



POLITECNICO
MILANO 1863

**SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE**

Design Document

COMPUTER SCIENCE AND ENGINEERING

Author(s): **Giacomo Orsenigo**
Federico Saccani
Francesco Ferlin

Deliverable:	DD
Title:	Design Document
Authors:	Giacomo Orsenigo, Federico Saccani, Francesco Ferlin
Version:	1.0
Date:	07-01-2024
Download page:	https://github.com/Furrrlo/FerlinOrsenigoSaccani/
Copyright:	Copyright © 2024, Giacomo Orsenigo, Federico Saccani, Francesco Ferlin - All rights reserved

Contents

Contents	i
1 Introduction	1
1.1 Scope	1
1.2 Definitions, Acronyms, Abbreviations	1
1.2.1 Definitions	1
1.2.2 Acronyms	1
1.2.3 Abbreviations	2
1.3 Revision history	2
1.4 Reference Documents	2
1.5 Document Structure	2
2 Architectural Design	4
2.1 Overview	4
2.2 Component view	6
2.2.1 Client Components	6
2.2.2 Server Components	6
2.2.3 Data Components	7
2.3 Deployment view	10
2.4 Runtime view	12
2.5 Component interfaces	26
2.6 Selected architectural styles and patterns	31
2.7 Other design decisions	32
3 User Interface Design	33
4 Requirements Treceability	37
5 Implementation, Integration and Test Plan	41
5.1 Development and Test Plan	42
5.2 Components integration	43
6 Effert Spent	47
Bibliography	48

List of Figures	49
List of Tables	51

1 | Introduction

1.1. Scope

The CodeKataBattle platform is a distributed software system that allows its users (students) to participate in challenges of different programming languages to improve their software development skills. Educators can create programming battles, providing a set of test cases that students must pass. Students can create teams and carry out the battles, providing their solutions using a test-driven approach. When a team submits a possible solution to the battle, the system runs tests and calculates the corresponding score.

The platform is implemented with the microservices architecture. More implementation choices are covered in later sections of this document.

1.2. Definitions, Acronyms, Abbreviations

1.2.1. Definitions

Slug: a human-readable and URL-friendly (as in, limited to lowercase ASCII characters) string which identifies a particular resource. It is typically used to identify resources in URLs, but can also be used as IDs instead of a more traditional way such as integers

GitHub Repository Slug: a slug with the structure ‘<repository owner>/<repository name>’ which uniquely identifies a repository across all of the ones hosted on GitHub. Can typically be seen in a repository URL

1.2.2. Acronyms

CKB: CodeKataBattle

API: Application Programming Interface

SAT: Static Analysis tools

GH: GitHub

SSO: Single Sign On

UUID: Universal Unique Identifier

DB: DataBase

DBMS: DataBase Management System

RPC: Remote Procedure Call

REST: REpresentational State Transfer

SPA: Single Page App

CDN: Content Delivery Network

1.2.3. Abbreviations

e.g.: For example

repo: Repository

ID: Identifier

1.3. Revision history

- **1.0** (7th January 2024) - Initial release

1.4. Reference Documents

Specification document: *"Assignment RDD AY 2023-2024"*

UML official specification: <https://www.omg.org/spec/UML/>

Requirements Analysis and Specification Document: *"RASD"*

1.5. Document Structure

1. Section 1: Introduction

This section gives a brief description of the problem and the scope of the system. It also contains the list of definitions, acronyms and abbreviations that might be encountered while reading the document. Additionally, there's the revision history of the document, which keeps track of the various version, their release date and their changes.

2. Section 2: Architectural Design

This section is about the architecture of the system. It firstly gives a high level overview of the architectural choices. Then it presents in details the components and deployment views, including server, client and data components (DB schemas). Additionally, it contains sequence diagrams that represent the runtime view of the system, as well as the component interfaces. Finally, this section focus on design choices, styles, patterns and paradigms

3. Section 3: User Interface Design

This section provides an overview on how the UI of the system will look like.

4. Section 4: Requirements Traceability

This section maps the requirements that have been defined in the RASD to the design elements defined in this document.

5. Section 4: Implementation, Integration and Test Plan

This section shows the order in which the subcomponents of the system will be implemented as well as the order in which subcomponents will be integrated and how to test the integration.

2 | Architectural Design

2.1. Overview

The system is a distributed application which follows the microservices architecture paradigm.

With the term client we are referring to either a Student or an Educator.

Users interact with the system by interfacing with a Single Page App (SPA), served by a Content Delivery Network (CDN). The website forwards appropriate requests to an API Gateway. The user is completely unaware of the microservices structure used to operate the system.

Each microservice realizes a service useful for the fulfillment of the single functionality that the CKB Platform is to provide. All requests go through the API Gateway which realizes complex services by sending requests to the several available microservices.

The CKB platform can be abstracted into 5 main areas:

1. Battles, Tournaments, Teams and Scores
2. Notifications
3. Badges
4. Building and testing
5. Analysing

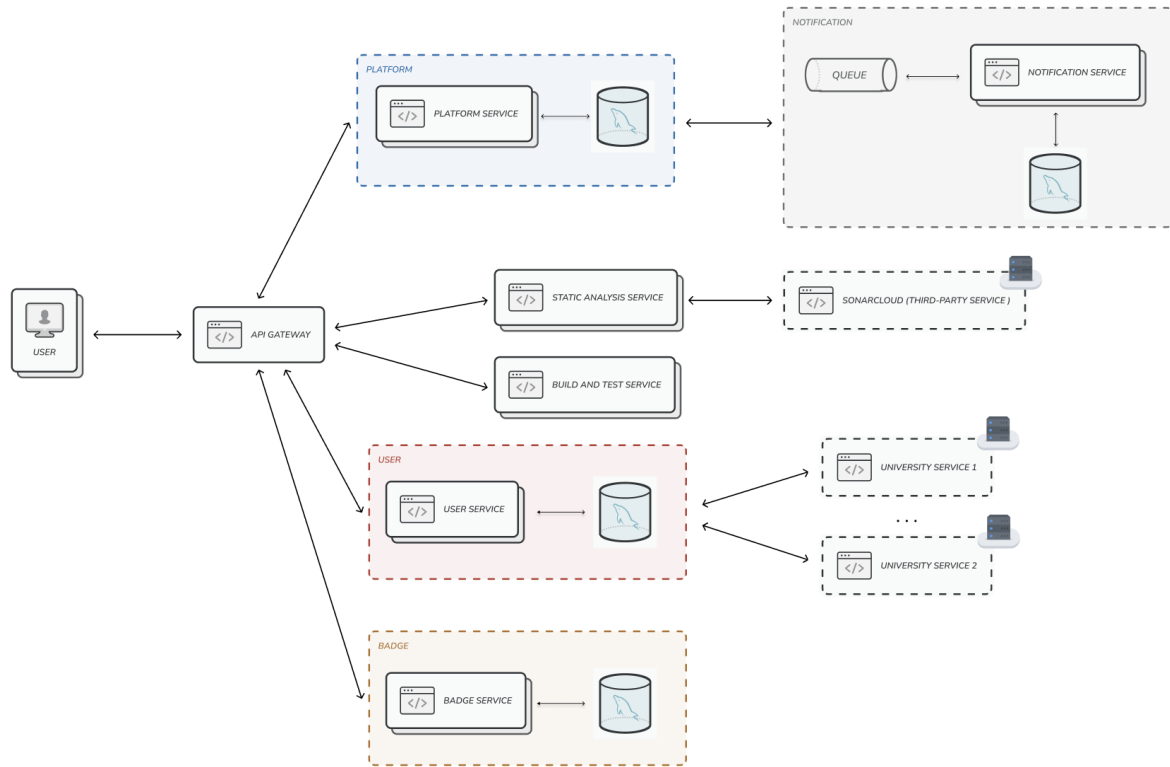


Figure 2.1: Abstract System View

These sections contain modules with functionality related to the same domain and independent of each other. Consequently, the areas will be implemented by different microservices ensuring low coupling and high cohesion.

The choice to use microservice architecture allows flexibility and scalability. All services can indeed be duplicated, since they don't have to interact with databases shared among microservices that could become bottlenecks of the system.

In addition, the components involved in sending notifications, compiling sources, running tests, and analyzing them perform more or less complex operations that may take a few moments of execution time. Consequently, they will be implemented in such a way as to be asynchronous, using of message queues to avoid making users wait for a long time (more details in later sections).

2.2. Component view

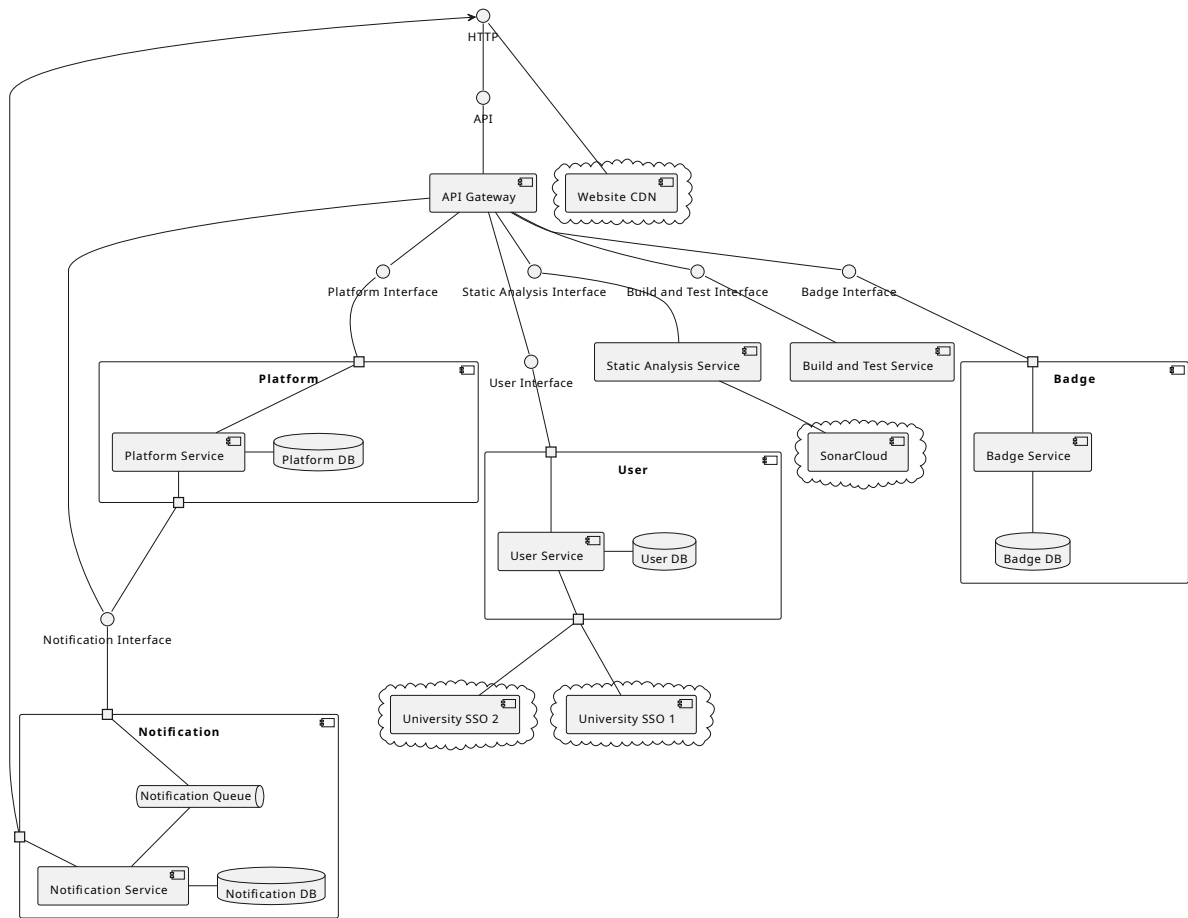


Figure 2.2: Component view diagram

2.2.1. Client Components

The frontend of the platform is a website that allows users to interact with the system. The role of the website is to interface the user with the API Gateway, rendering interfaces and requesting data.

2.2.2. Server Components

The server components contain the business logic needed to provide the functionalities of the application to the clients, by responding to requests made by Client Components as well as perform tasks based on interaction with the external platforms used by the clients (such as GitHub). There are 5 main components, which corresponds to the main areas identified above, as well as a few additional components:

- **User Service** - implements the authentication as well as keeps track of both students and educators by making use of 3rd-party authentication services provided by the institutions.
- **Platform Service** - implements all the logic related to creating and managing

battles and tournaments, including calculating the score and keeping track of the leaderboards

- **Badge Service** - implements all the logic related to the evaluation and assignment of badges, including the gathering of information needed in order to do that (for example Git or GH metadata such as number of commits by students, etc.) as well as the actual evaluation of JavaScript code which gets used by educators in order to express the logic of specific badges
- **Build and Test Service** - implements the Continuous Integration aspect, by building students projects and running them against tests for each push
- **Static Analysis Service** - abstracts away the interaction with SonarCloud, the 3rd party service which will perform the actual static analysis. While SonarCloud already exposes their own API for interaction, it is cumbersome (as it requires registering a webhook [12], importing the project from the GH slug [10], specifying the required analysis metrics in a quality profile [11] and waiting for a response on the hook). Therefore, by abstracting it away we can use it much more easily and, additionally, make it much less inconvenient to replace it should the need arise.
- **Notification Service** - responsible for delivering push notifications to client devices using external notification APIs (such as the Web Push API [16])
- **Website CDN** - responsible for serving the static files of the SPA to the clients
- **API Gateway** - responsible for connecting together all microservices, it is the component that provides a REST interface externally, which the clients will send requests to, which will be implemented by a series of internal calls using gRPC [4] to all the other microservices in order to make responses. The component does not implement a specific functionality per se, but is in charge of providing an interface to the clients as well as enforcing the correct usage of said interface.

2.2.3. Data Components

Server Components make use of 4 distinct DBMSes, each with its own schema.

- **User DB** - saves data related to the users of the platform. Its ER schema is not shown here as it would depend on the 3rd-party service, but it would at least require for each student and educator a unique identifier, as well as human-readable identifier to allow users to look up each other (so name and surname or an email). Educators and Students are saved in different tables.

- Platform DB

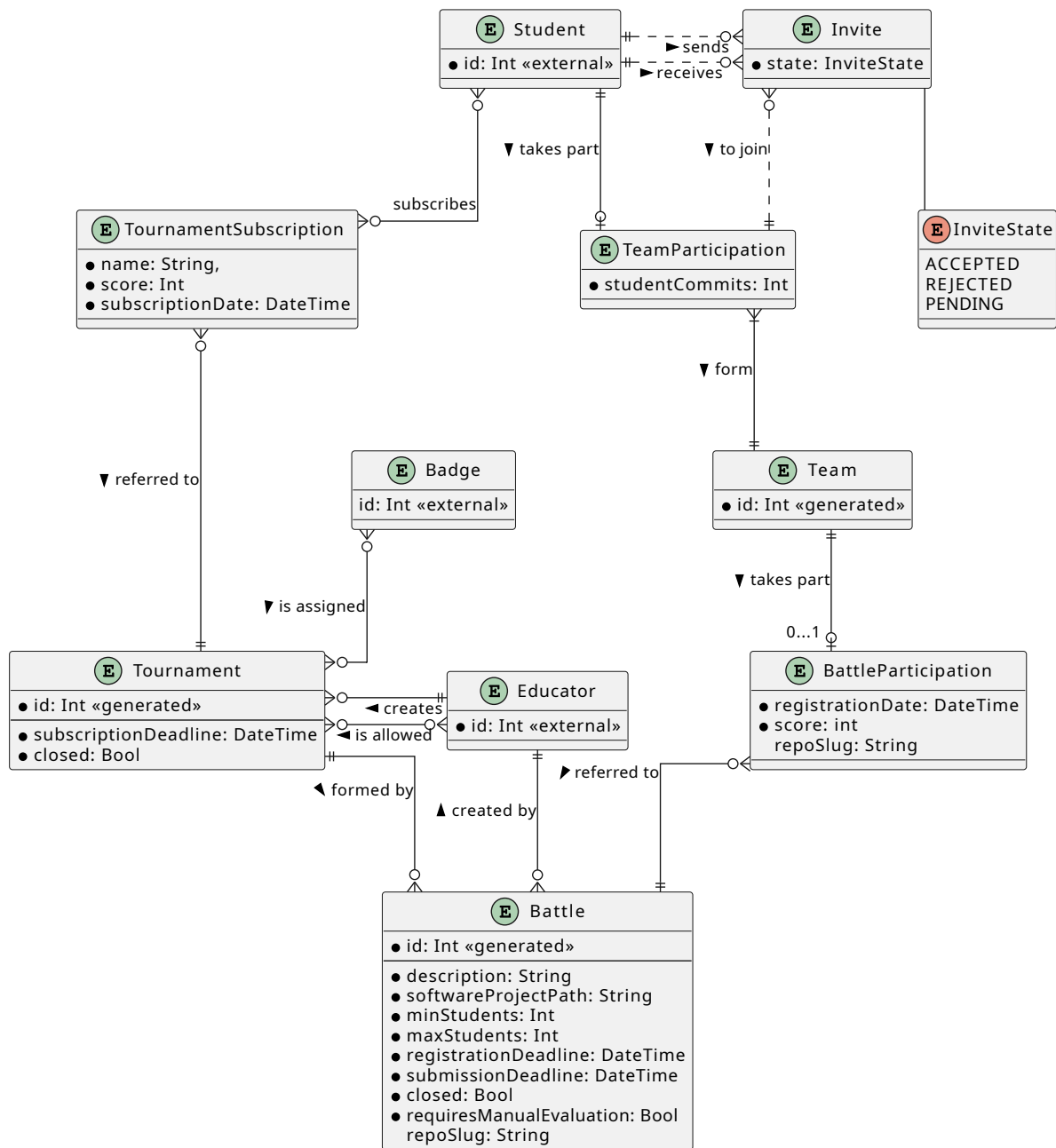


Figure 2.3: Platform ER

- Badge DB

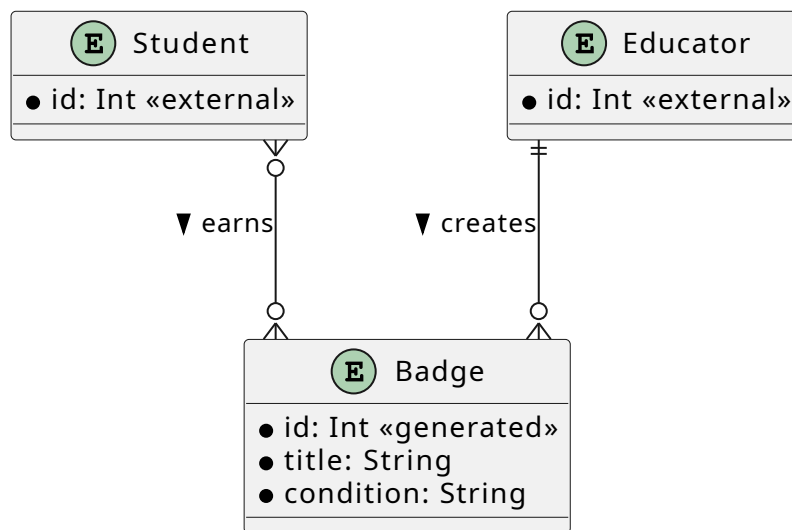


Figure 2.4: Badges ER

- Notification DB

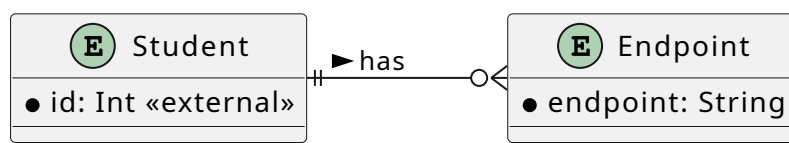


Figure 2.5: Notification ER

2.3. Deployment view

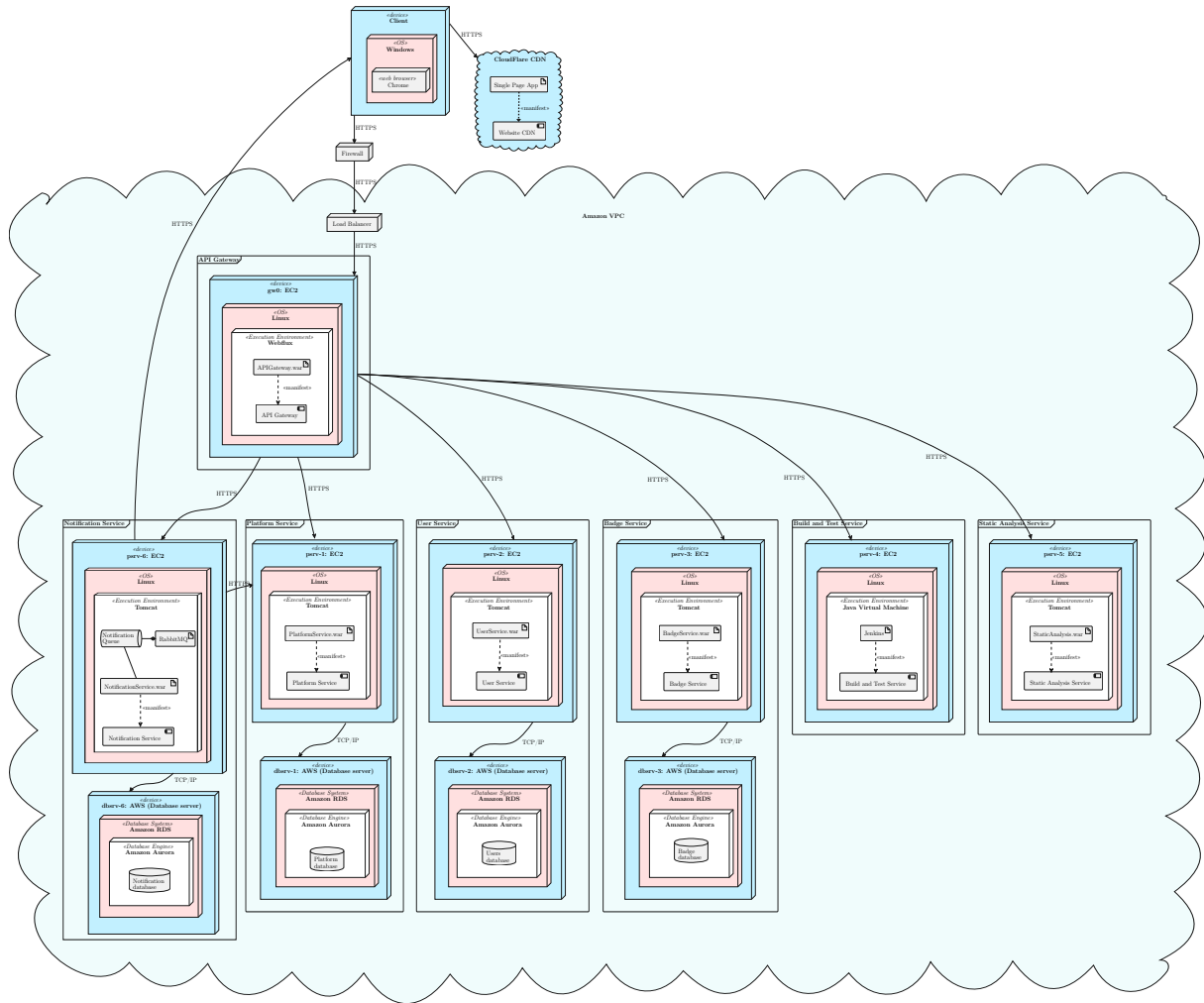


Figure 2.6: Deployment view diagram

The frontend is a SPA implemented with the React framework. The website can be visited from any modern web browser, such as Google Chrome, Mozilla Firefox or Microsoft Edge. Clients can be any device that can run the previous browsers, such as computers with Windows, Linux or macOS and smartphones/tablets with Android or IOS.

The website is served by a CDN, such as Cloudflare, Amazon CloudFront or Akamai. This allows to always have the maximum efficiency, regardless of the number of users connected at the same time.

All other microservices are protected by a firewall that allows only HTTPS connections to the API Gateway. All microservices can be hosted on-premise or in cloud, within a virtual network, such as Amazon VPC [3]. Since the expected workload will not be constant over time (all students study more or less at the same times), an on-premise architecture would be unused most of the time, resulting in a waste of money. The best choice is therefore a cloud architecture, such as Amazon EC2 [1], that allows to scale up resources only when really needed.

The API Gateway is stateless and can be implemented with the Spring Cloud Gateway [15]. It is built on top of the Reactor project [8], which provides a non-blocking API for routing and filtering requests. This allows to handle high traffic and scale the API Gateway horizontally when necessary. The API Gateway is also responsible for performing load balancing in case other microservices are replicated.

If the API Gateway is replicated, a load balancer, such as the Amazon ELB [2], is necessary to equally distribute requests among the various instances.

The Build and Test Service is based on the Jenkins system [6]. It is the most CPU-intensive service and should adopt a distributed builds architecture [7]. One possibility is to have a single Jenkins controller that schedules builds to dedicated build nodes (agents). Build agents can be dynamically deployed to a new EC2 instance and removed, based on the current workload.

If the CKB platform will grow up, an architecture with multiple controllers can be adopted. For example, the platform can create a Jenkins controller for each tournament.

All other service are implemented as Spring Boot [14] applications and they can be deployed to EC2 instances.

The Platform Service, Badge Service, User Service and Notification Service also have their own database. They use a managed DB service, such as Amazon RDS, that can automatically manage backups and resource scaling. Each database is in a private subnetwork and it can be accessed only by its corresponding service.

The Notification Service uses RabbitMQ [9] to create a queue where the Platform Service can push messages. The service will periodically pop messages from the queue with the Spring AMQP project [13] and send them.

2.4. Runtime view

The following sequence diagrams represent the dynamics of interaction between components.

Sequence diagrams from RW1 to RW9 are the realizations of the corresponding use cases in the RASD document.

Sequence diagram RW0.1 represents the login process.

Sequence diagrams from RW0.2 to RW0.5 are common part that have been extracted for simplicity and are not shown in other diagrams.

In the first four sequence diagrams, the user could be a student or an educator.

RW0.1:

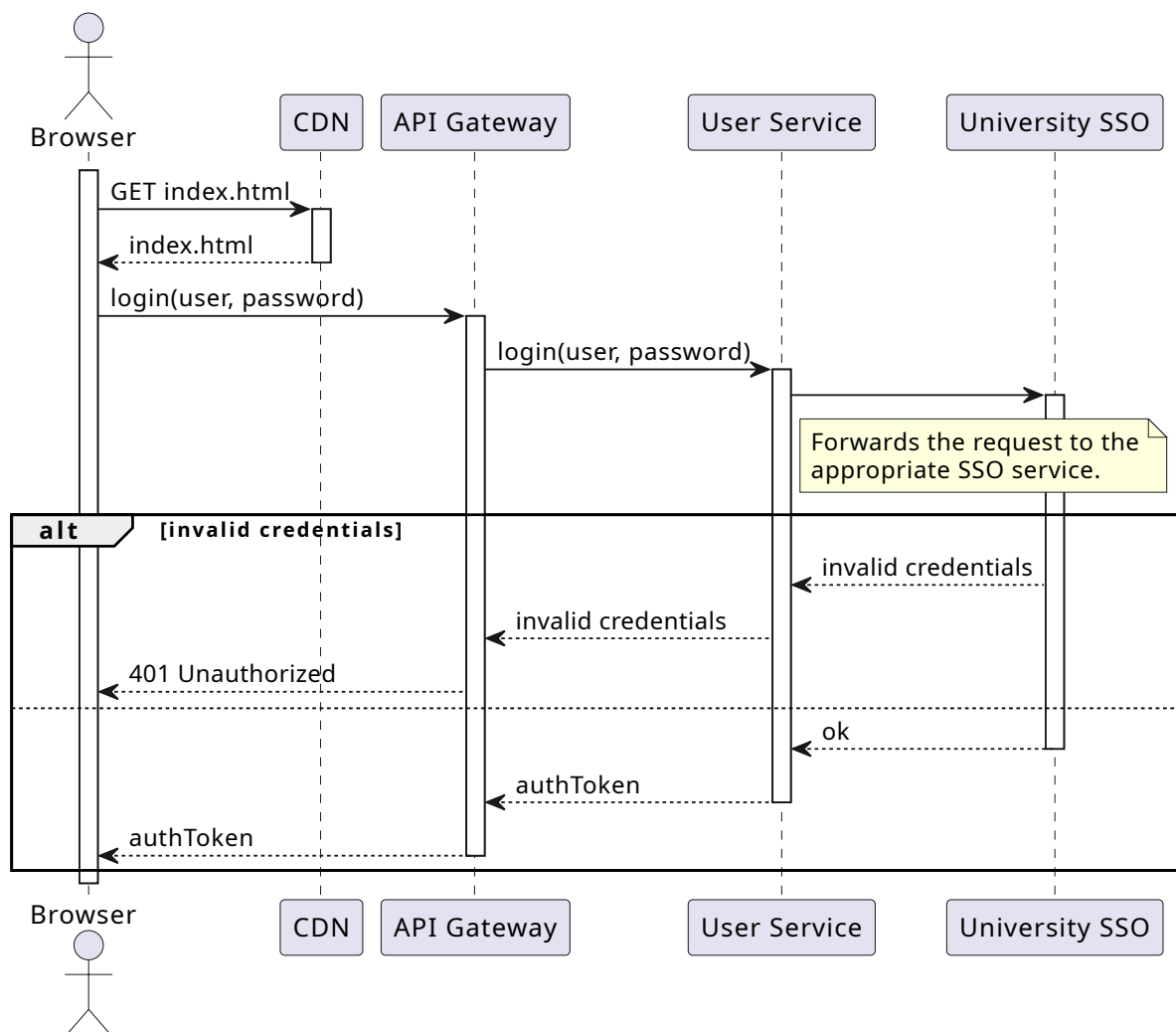


Figure 2.7: Login

When the User Service receives the login request, it contacts the external SSO service to validate it. If the credential provided by the user are correct, the User Service returns an authToken, that must be included in all subsequent requests.

RW0.2:

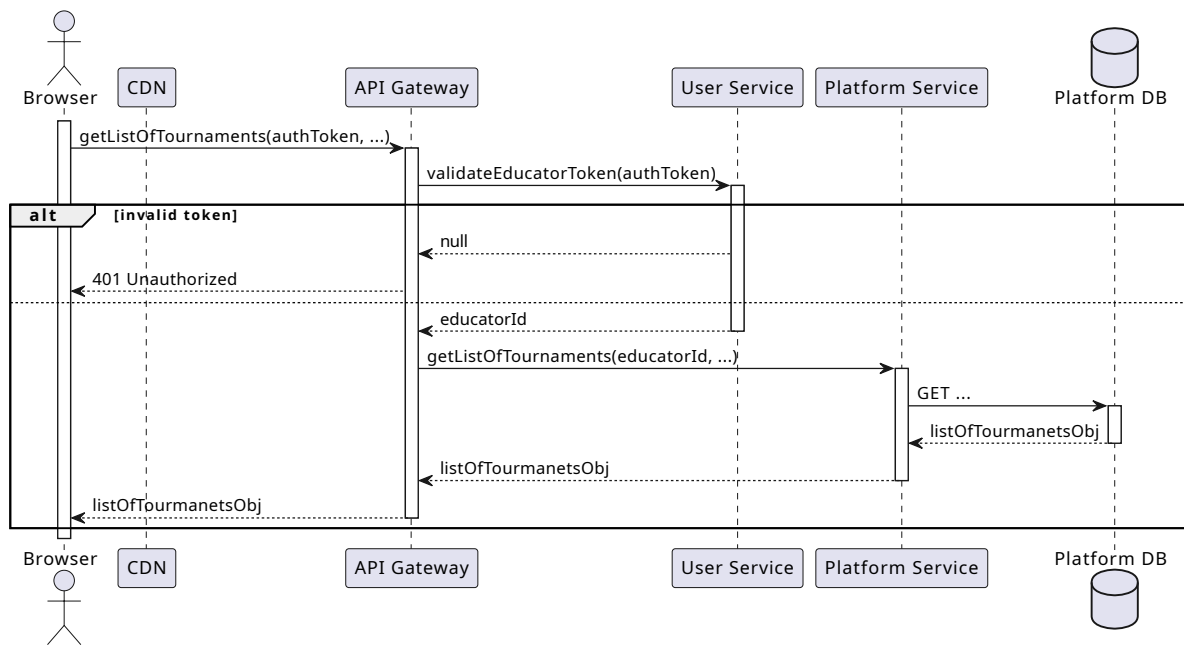


Figure 2.8: User gets list of tournaments

RW0.3:

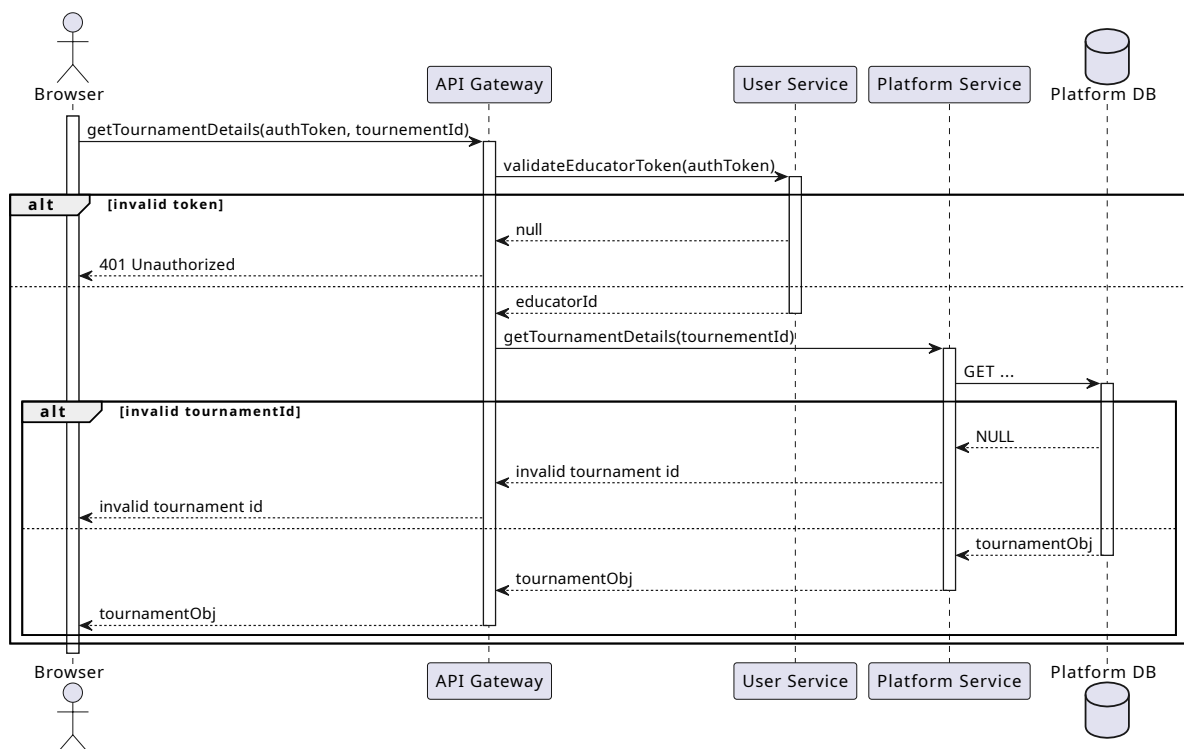


Figure 2.9: User gets tournament details

RW0.4:

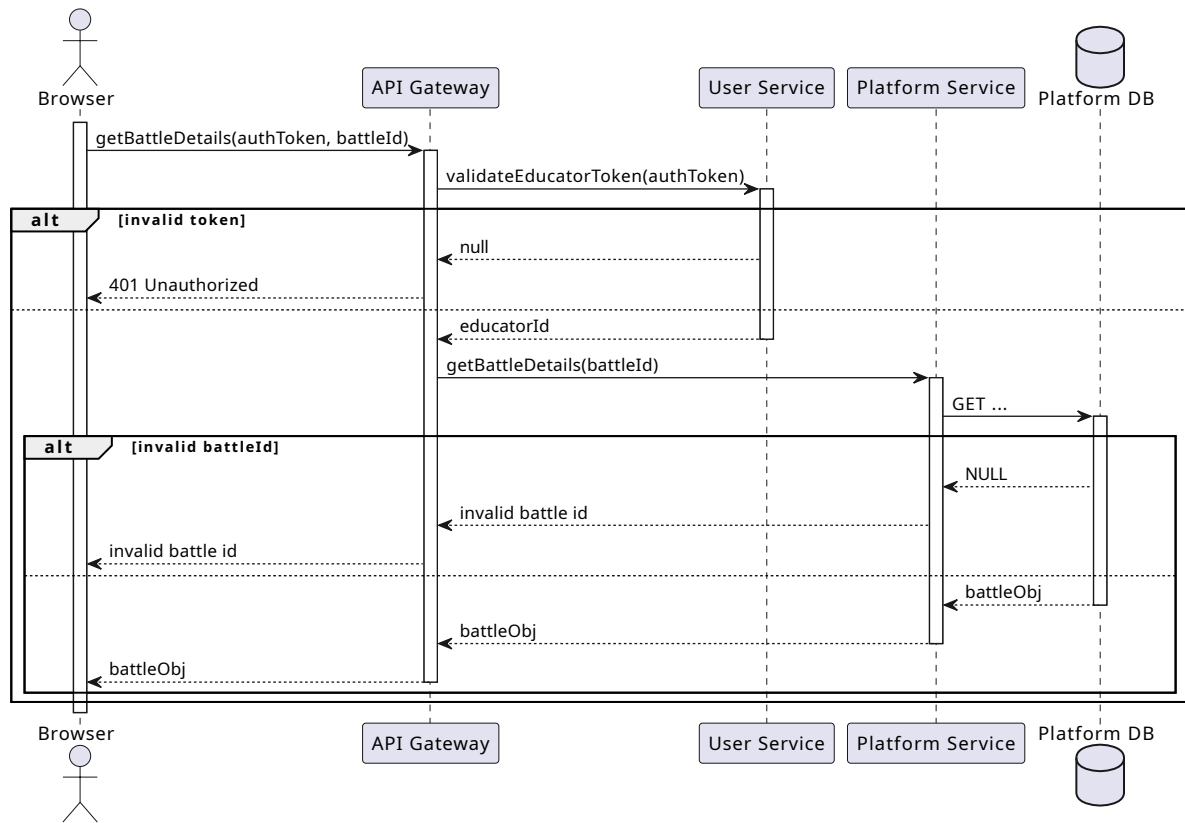


Figure 2.10: User gets battle details

RW0.5:

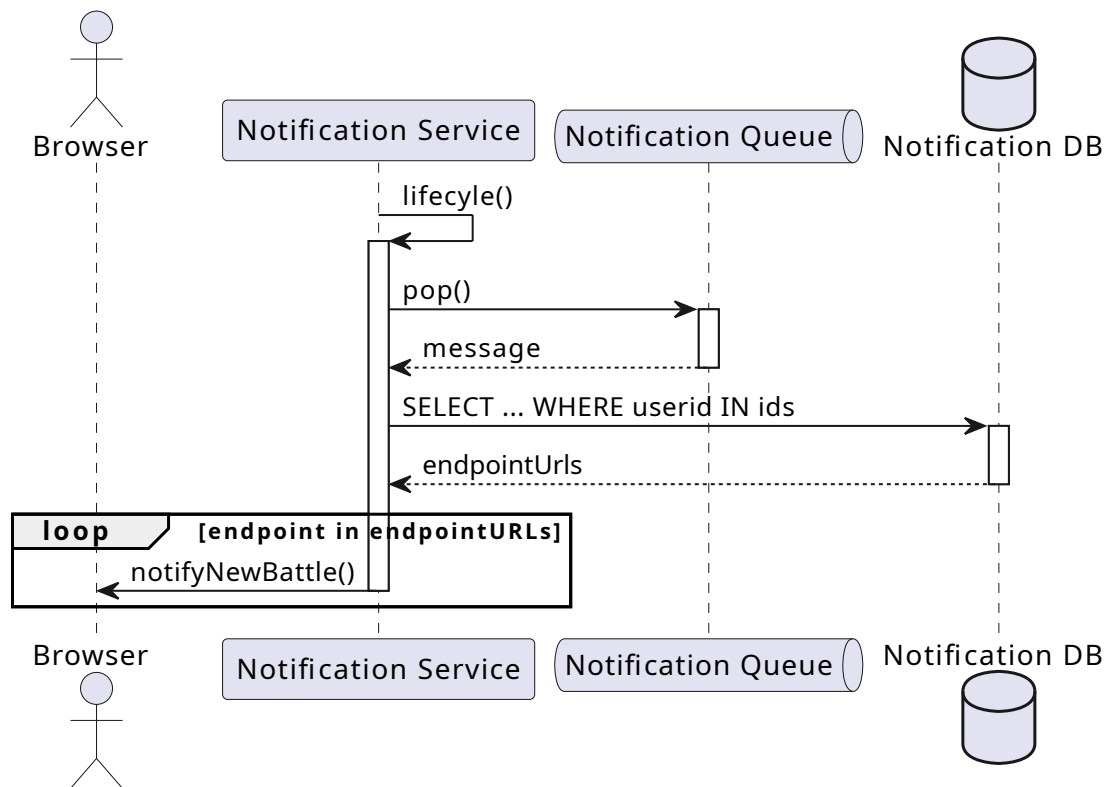


Figure 2.11: Notification is sent

This diagram shows the process of sending a notification. The Notification Service will periodically pop from the Notification Queue messages that contain an array of IDs of the recipient students. Then the Notification Service will look for each student's endpoint URL (previously registered by clients using the API Gateway) and sends the notifications.

RW1:

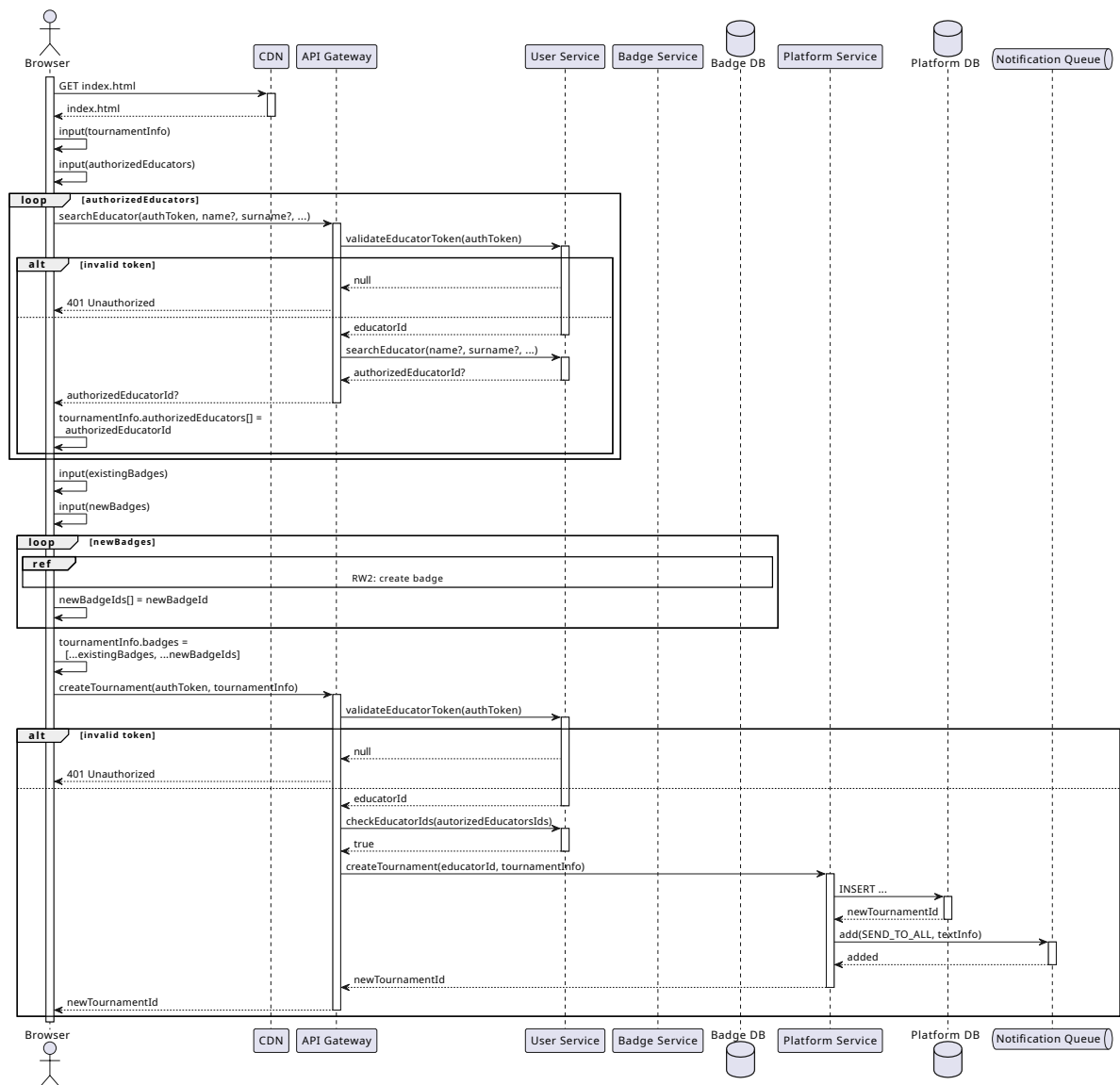


Figure 2.12: Educator creates a new Tournament

RW2:

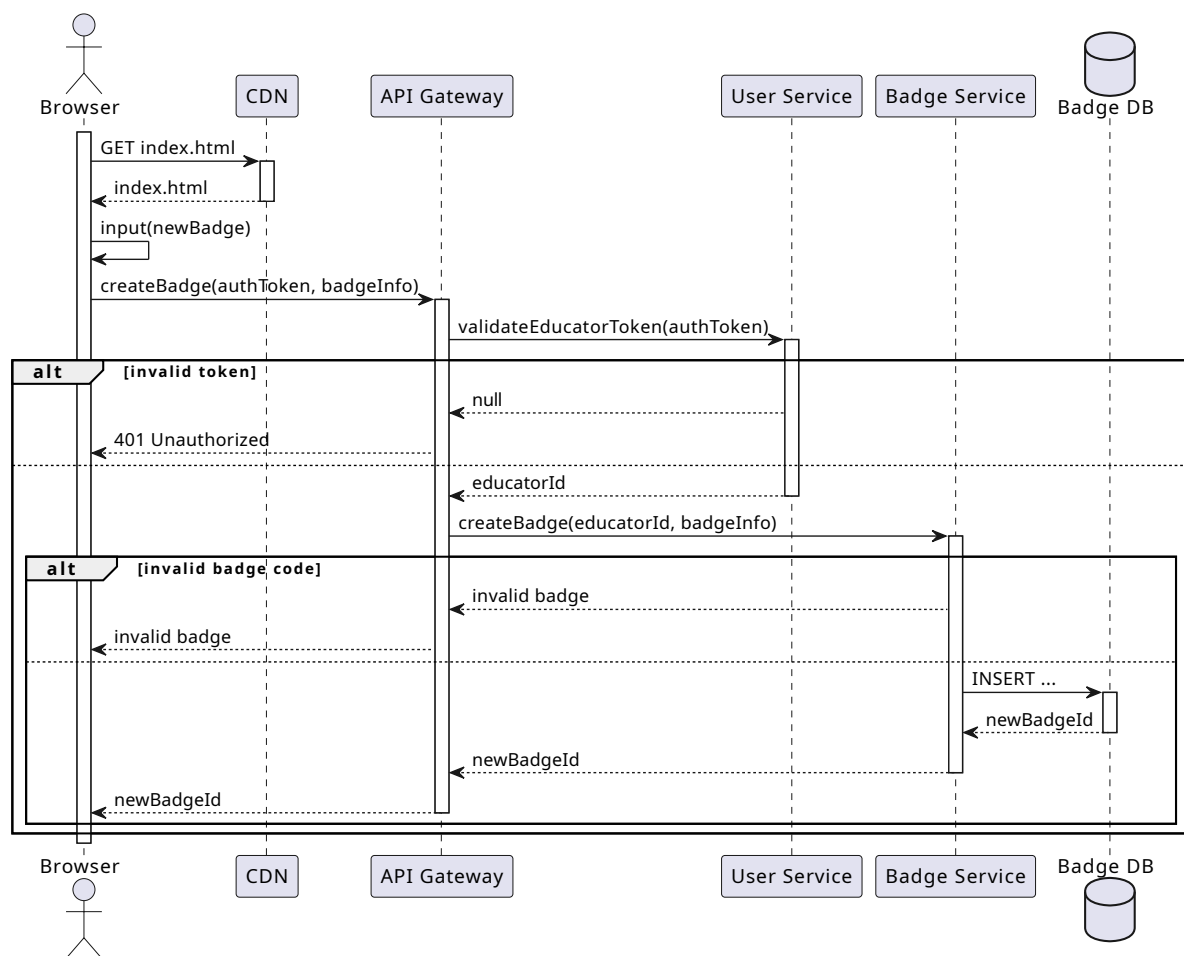


Figure 2.13: Educator creates a new badge

RW3:

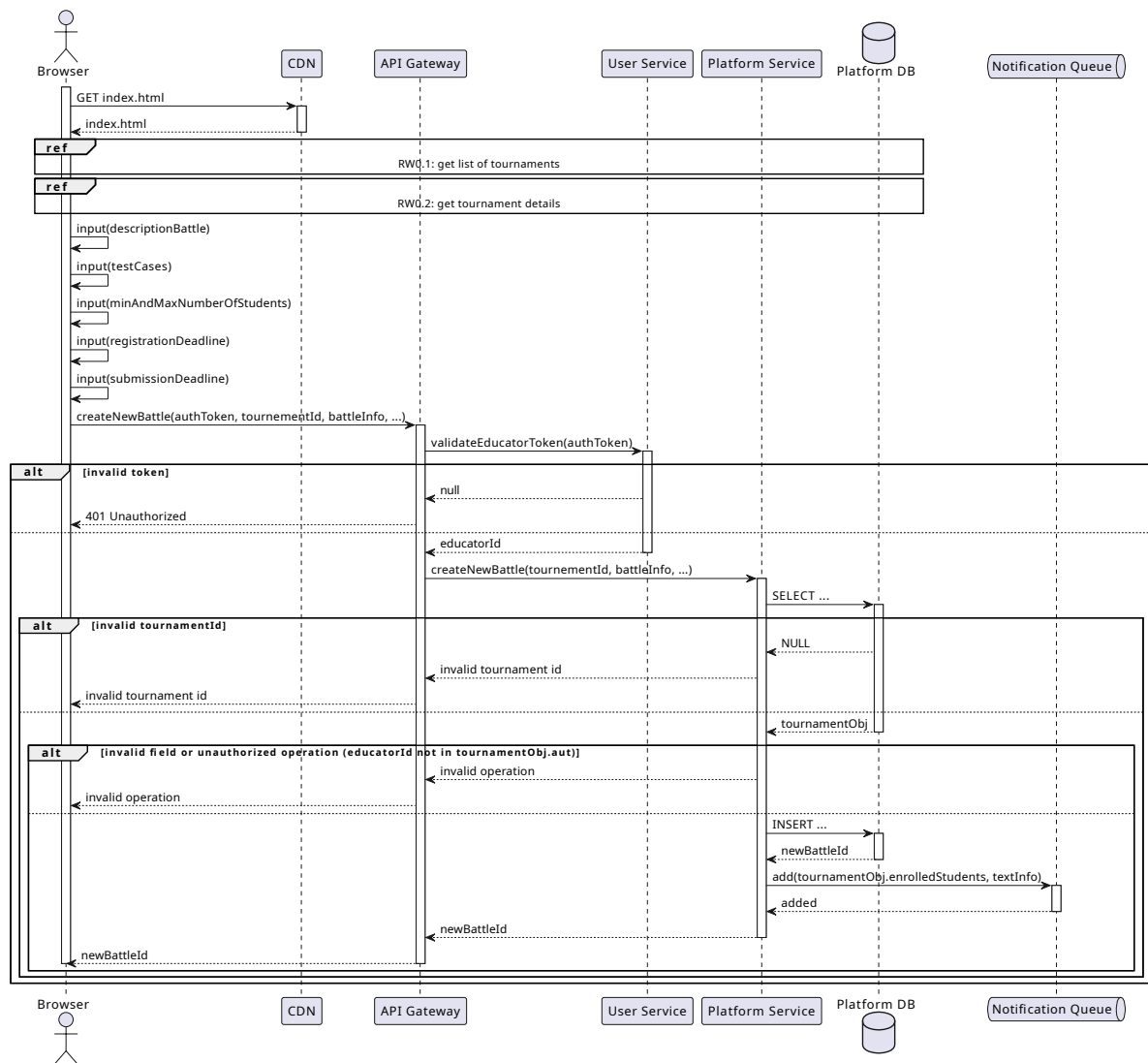


Figure 2.14: Educator creates a new Battle for an Existing Tournament

RW4:

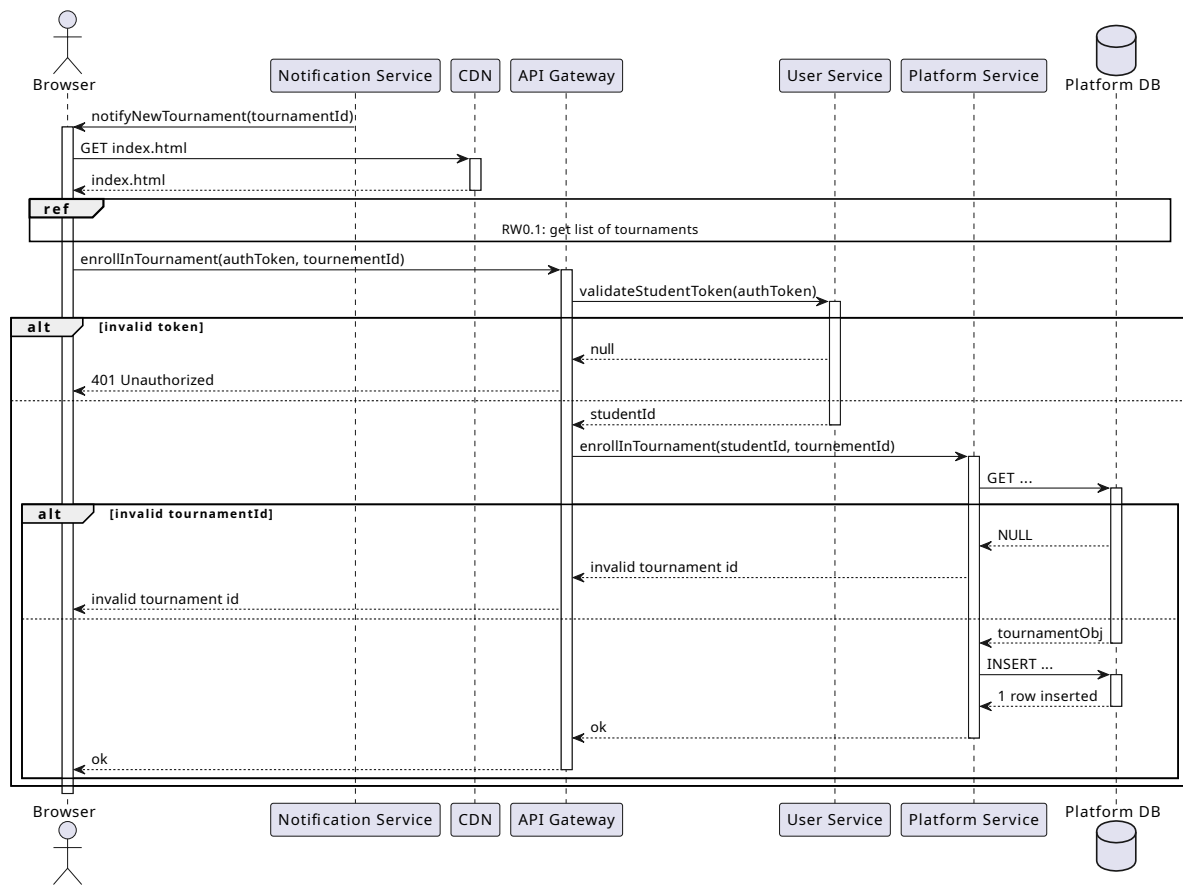
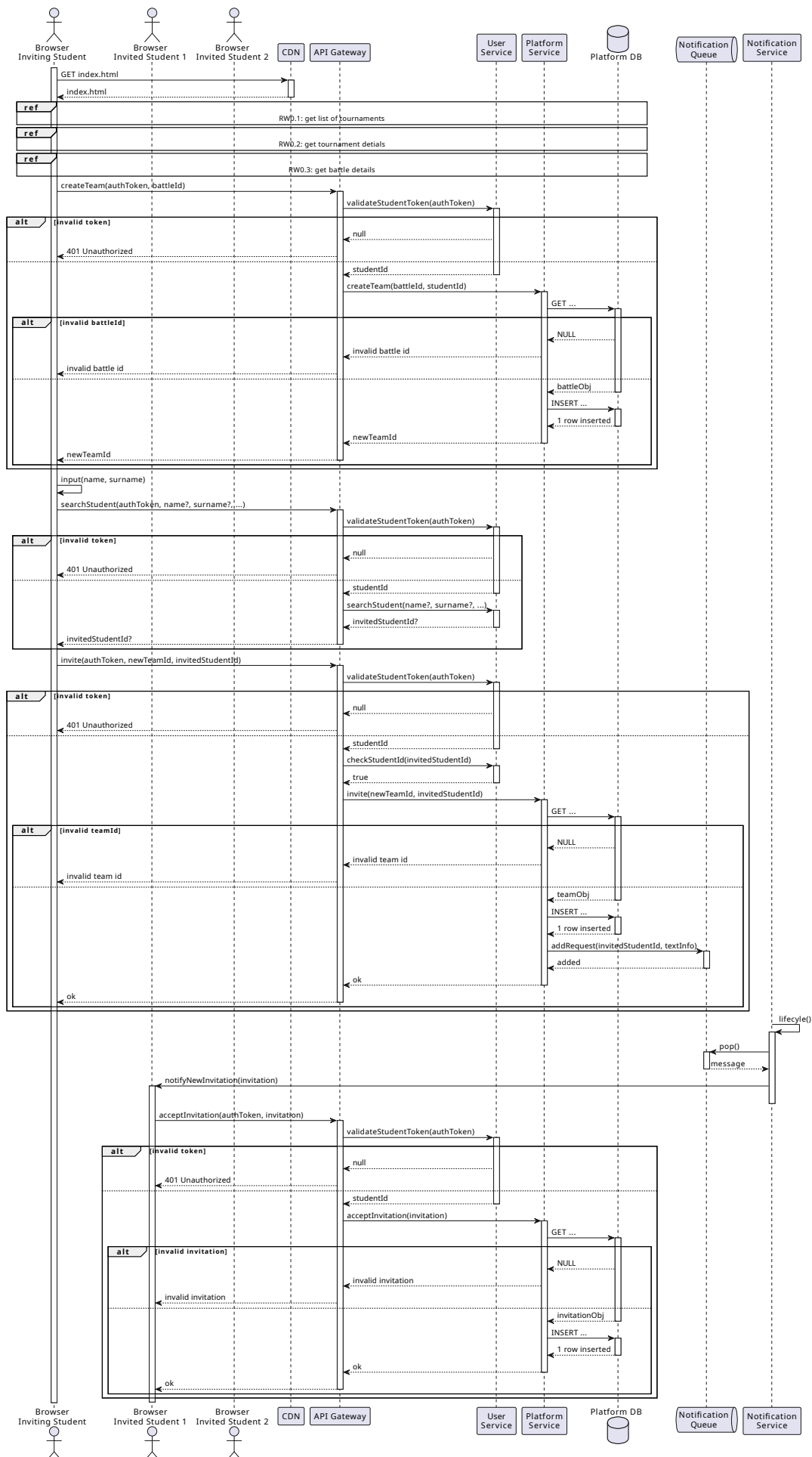


Figure 2.15: Student joins to an existing Tournament by receiving a notification

RW5:



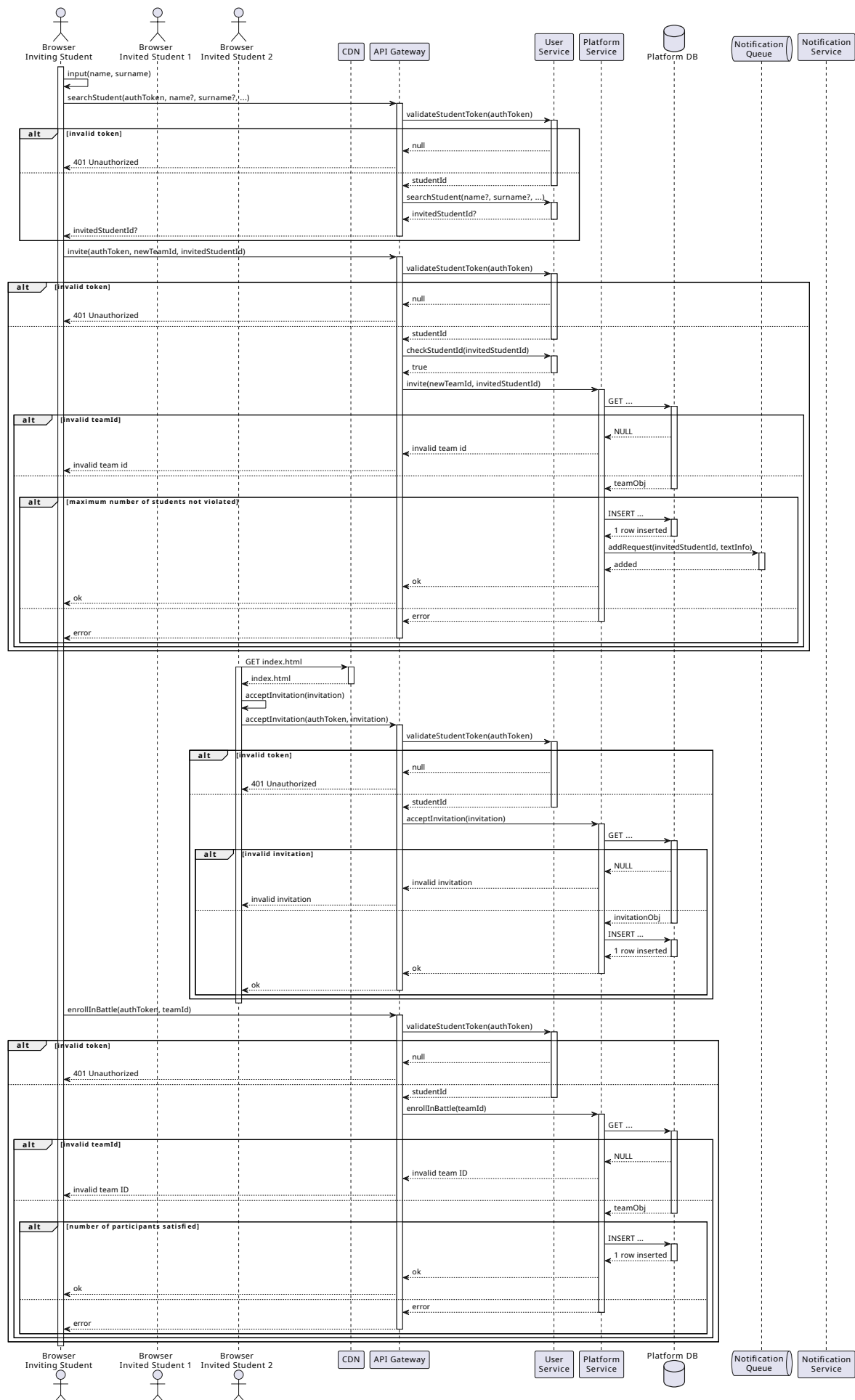


Figure 2.16: Students create a team for a tournament battle

RW6:

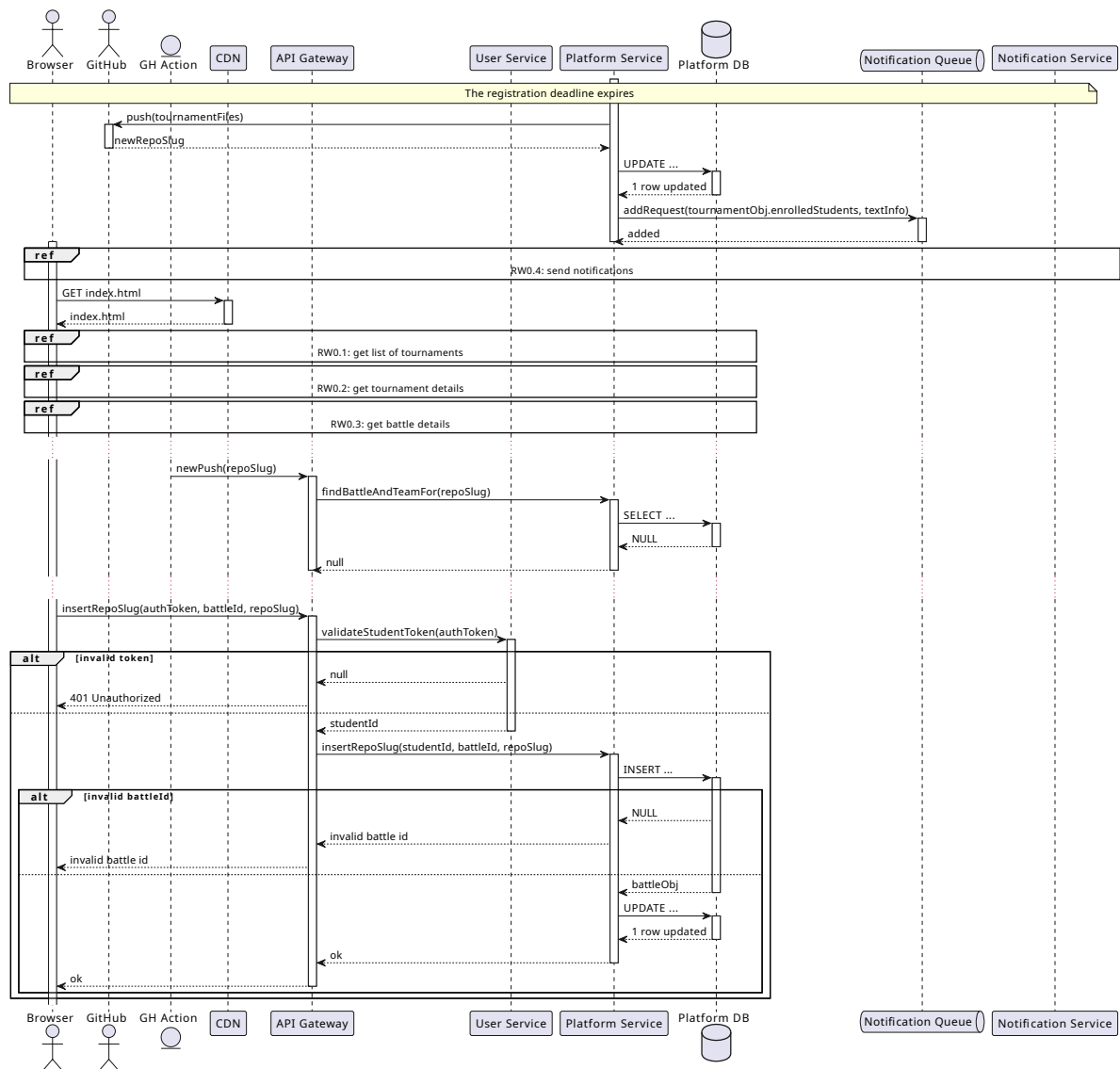


Figure 2.17: Student forks the repository

RW7:

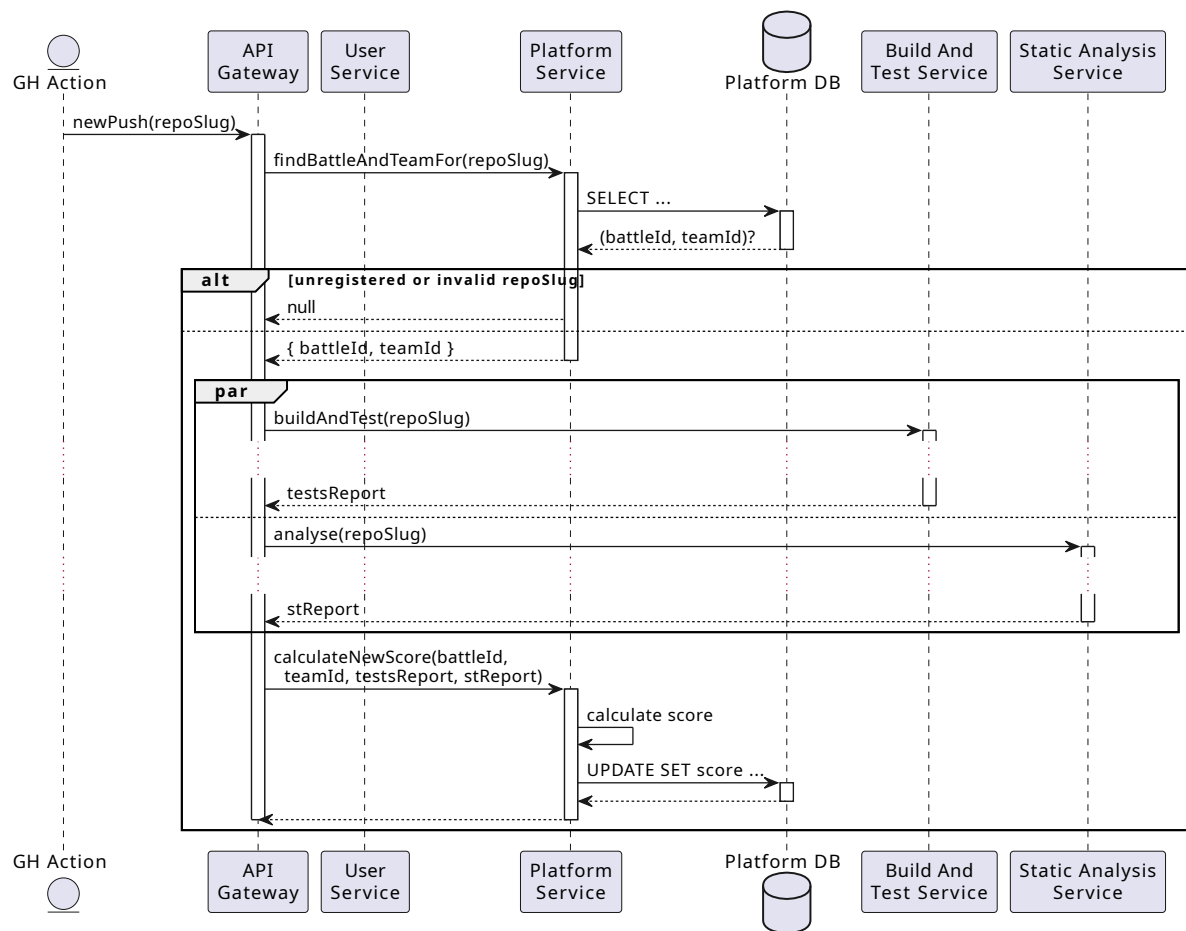


Figure 2.18: Student pushes and triggers automatic evaluation

RW8:

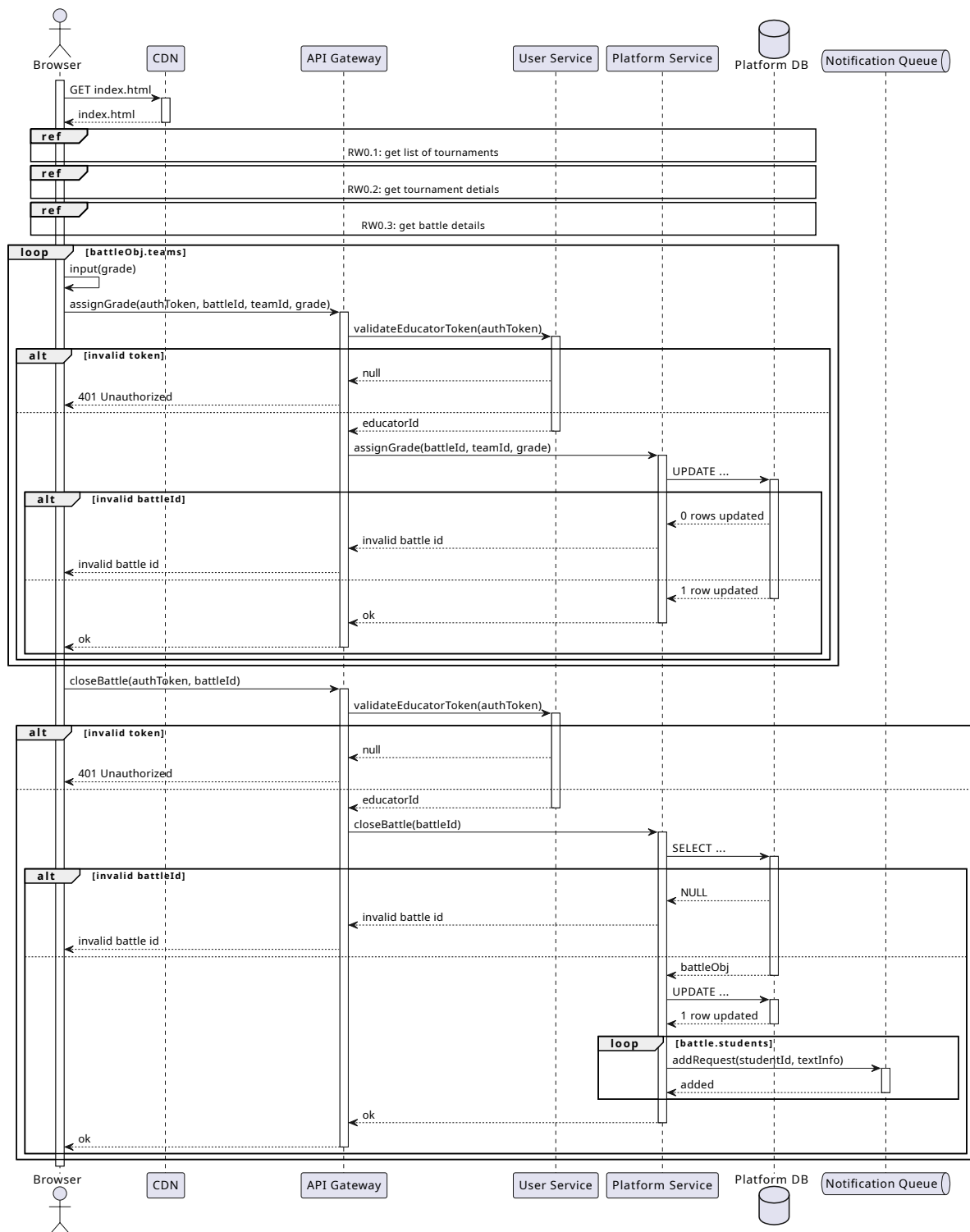


Figure 2.19: Educator manually evaluates teams

RW9:

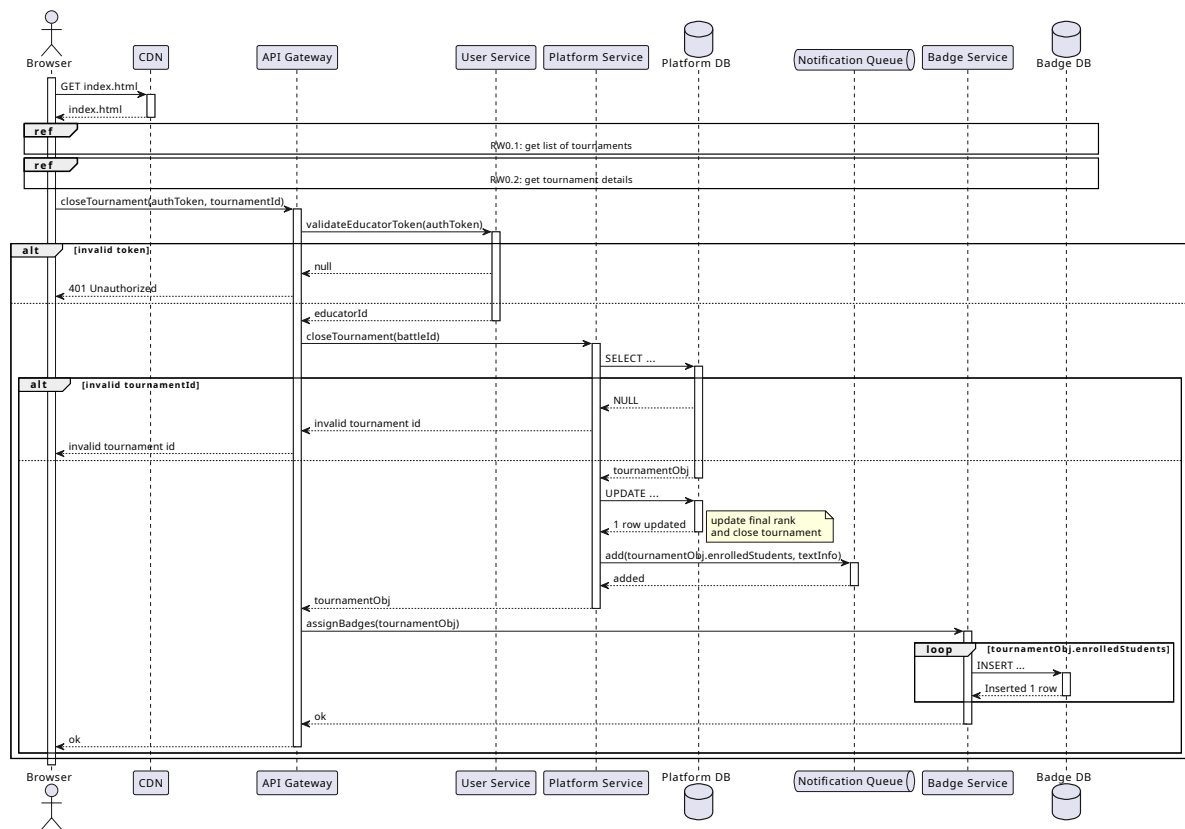


Figure 2.20: Educator closes a tournament

2.5. Component interfaces

Here the most relevant interfaces exposed by components are described, including all the operations seen in the previous diagrams:

- **API Gateway**

- **login(username: String, password: String): String?** logs the user in through the sso service and, if successful, returns the authToken
- **getListOfTournaments(authToken: String): List<SimpleTournament>** given a valid educator auth token, returns the list of tournaments which he has created or he has permissions to add battles to
- **getTournamentDetails(authToken: String, tournamentId: ID): Tournament** returns the detailed tournament info for the given id
- **getBattleDetails(authToken: String, battleId: ID): Battle** returns the detailed battle info for the given id
- **searchEducator(authToken: String, name: String?, surname: String?, ...): ID?** search for an educator matching the given name and/or surname
- **searchStudent(authToken: String, name: String?, surname: String?, ...): ID?** search for a student matching the given name and/or surname
- **createTournament(authToken: String, tournamentInfo: NewTournament): ID** create a new tournament with the given info and return its id
- **createBadge(authToken: String, badgeInfo: NewBadge): ID** create a new badge with the given info and return its id
- **createNewBattle(authToken: String, tournamentId: ID, battleInfo: Battle): ID** create a new battle with the given info and return its id
- **enrollInTournament(authToken: String, tournamentId: ID): void** enroll the student owning the given token in the given tournament
- **createTeam(authToken: String, battleId: ID): ID** create a new team to take part in the given battle and assign the student owning the given token to it, returning the id of the created team
- **invite(authToken: String, newTeamId: ID, invitedStudentId: ID): void** send an invitation for the given team to the invitedStudentId, with the student owning the given token as the invite sender
- **acceptInvitation(authToken: String, invitation: ID): void** accept the given invitation

- **rejectInvitation(authToken: String, invitation: ID): void** reject the given invitation
- **enrollInBattle(authToken: String, teamId: ID): void** enroll the given team in the battle
- **insertRepoSlug(authToken: String, battleId: ID, repo: Slug): void** associate the given GH repository to the team of the student owning the given token for the specified battle
- **newPush(repo: Slug): void** triggered by the GHA on a new push to the specified repository
- **assignGrade(authToken: String, battleId: ID, teamId: ID, grade: Int): void** assign this grade to the submission of the specified team in the given battle
- **closeTournament(authToken: String, tournamentId: ID): void** close the given tournament
- **closeBattle(authToken: String, battleId: ID): void** close the given battle while it's in its consolidation phase
- **addStudentNotificationEndpoints(authToken: String, endpoint: String): void** add the URL of the notification endpoints of a student

- **User Interface**

- **login(username: String, password: String): String?** logs the user in through the sso service and, if successful, returns the authToken
- **validateEducatorToken(authToken: String): ID?** validate that the given token is a valid educator token and return its educator id
- **validateStudentToken(authToken: String): ID?** validate that the given token is a valid student token and return its student id
- **searchEducator(name: String?, surname: String?, ...): ID?** search for an educator matching the given name and/or surname
- **searchStudent(name: String?, surname: String?, ...): ID?** search for a student matching the given name and/or surname
- **checkEducatorIds(educatorsIds: ID[]): Bool** checks that the given ids are all associated to an existing educator
- **checkStudentId(studentId: ID): Bool** checks that the given id is associated to an existing student

- **Platform Interface**

- **getListOfTournaments(educatorId: ID): List<SimpleTournament>**
given an educator id, returns the list of tournaments which he has created or he has permissions to add battles to
- **getTournamentDetails(tournamentId: ID): Tournament** returns the detailed tournament info for the given id
- **getBattleDetails(battleId: ID): Battle** returns the detailed battle info for the given id
- **createTournament(creatorId: ID, tournamentInfo: NewTournament): ID** create a new tournament with the given info and return its id
- **createNewBattle(tournamentId: ID, battleInfo: NewBattle): ID** create a new battle with the given info and return its id
- **enrollInTournament(studentId: ID, tournamentId: ID): void** enroll the given student in the given tournament
- **createTeam(battleId: ID, studentId: ID): ID** create a new team to take part in the given battle and assign the student owning the given token to it, returning the id of the created team
- **invite(newTeamId: ID, invitedStudentId: ID): void** send an invitation for the given team to the specified student, with the student owning the given token as the invite sender
- **acceptInvitation(invitation: ID): void** accept the given invitation
- **rejectInvitation(invitation: ID): void** reject the given invitation
- **enrollInBattle(teamId: ID): void** enroll the given team in the battle
- **insertRepoSlug(studentId: ID, battleId: ID, repo: Slug): void** associate the given GH repository to the team of the student owning the given token for the specified battle
- **findBattleAndTeamFor(repo: Slug): battleId: ID, teamId: ID?**
- **calculateNewScore(battleId: ID, teamId: ID, testsReport: TestReport, stReport: SatReport): void**
- **assignGrade(battleId: ID, teamId: ID, grade: Int): void** assign this grade to the submission of the specified team in the given battle
- **closeTournament(tournamentId: ID): void** close the given tournament
- **closeBattle(battleId: ID): void** close the given battle while it's in its consolidation phase

- **Badge Interface**

- **createBadge(creatorId: ID, badgeInfo: NewBadge): ID** create a new badge with the given info and return its id
- **createVariable(name: String, code: String): void** create a new variable that can be used in badges
- **assignBadges(tournament: Tournament): void** evaluate and assign badges to all the students that took part in the given tournament
- **Notification Interface**
 - **addStudentNotificationEndpoints(studentId: ID, endpoint: String): void** add the URL of the notification endpoints of a student
- **Static Analysis Interface**
 - **analyse(repo: Slug): SatReport** run static analysis on the given repository and produce a report
- **Build and Test Interface**
 - **buildAndTest(repo: Slug): TestReport** build the given repository, run its tests and produce a report

Types:

- **ID** - a unique identifier for a given object, could be a self-incrementing integer or a UUID (depends on the chosen DBMS)
- **Slug** - a String identifying a specific GitHub repository, in the format ‘<user-name>/<repository>’ as can be seen in a GitHub project URL
- **SimpleTournament**

```
{
    id: ID,
    name: String,
    status: Subscribing | InProgress | Closed,
}
```
- **Tournament**

```
{
    id: ID,
    name: String,
    status: Subscribing | InProgress | Closed,
    subscriptionDeadline: DateTime,
    enrolledStudents: { id: ID, name: String, score: Int }[],
    battles: { id: ID }[],
}
```

- **NewTournament**

```
{
  name: String ,
  subscriptionDeadline: DateTime,
  authorizedEducators: { id: ID },
  badges: { id: ID }[],
}
```

- **Battle**

```
{
  name: String ,
  description: String ,
  status: Registration | Submission | Consolidation | Done,
  minStudents: Int ,
  maxStudents: Int ,
  registrationDeadline: DateTime,
  submissionDeadline: DateTime,
  requiresManualEvaluation: Bool,
  repo: Slug?,
  teams: { id: ID, students: { id: ID }[], score: Int }[],
}
```

- **NewBattle**

```
{
  name: String ,
  description: String ,
  minStudents: Int ,
  maxStudents: Int ,
  registrationDeadline: DateTime,
  submissionDeadline: DateTime,
  requiresManualEvaluation: Bool,
  zippedProject: Blob ,
  privateProjectFiles: Path[] ,
  sonarRules: {
    key: String ,
    parameters: {
      key: String ,
      value: String ,
    }[],
  }[],
}
```

The `zippedProject` field is a binary blob which contains all project files and tests, both public and private. The `privateProjectFiles` is in charge of defining which files from this zip will be hidden from the students, so that educators can specify private tests. Care must be taken by educators so that the project build script can also work in the absence of those files. The `sonarRules` field lists all the rules to be applied by SonarCloud when scanning.

- **NewBadge**

```
{
    title: String,
    condition: String,
}
```

- **TestReport**

```
{
    passed: Int,
    failed: Int,
}
```

- **SatReport**

```
Map<{ metricName: String }, { score: Int }>
```

2.6. Selected architectural styles and patterns

Fat client: fat clients allow offering a wide variety of functionalities independent from the central server, as well as to move part of the business logic off of the server and into the clients. The main advantages it offers are greater decoupling of frontend and backend as well as a better interactive experience, especially in conditions where the client is on an unstable network. Additionally, with the adoption of a single-page applications, there are many cross-platform frameworks which allow to reuse code partially or entirely on multiple platforms. The single-page web application can also be served by a dedicated static webserver, as all its interactivity is implemented client-side, further reducing the burden on the server by delegating its serving to a CDN, which is highly optimized for this specific use case.

REST API: an architectural style defined on top of HTTP centered around the definition of a standardized set of stateless operations. Its main advantages are its simplicity, its use of widely adopted standards which facilitate adoption, and its ease of scalability given by its stateless nature. It allows to have a single API interface against which a heterogeneous set of clients can make requests.

Microservices architecture: an architectural style where a single application is developed as a suite of small services, each running in its own process and each dealing with

a single bounded context in the target domain. The use of this architecture allows us a more fine-grained scaling strategy, as we can clearly select in our system CPU-bound services (such as the Build and Test Service) which will need to have many more resources allocated to it compared to the rest of the system.

API Gateway: a mediator which sits between clients and a service being invoked, which allows to implement cross-cutting concerns such as authentication mechanism common for all microservices. It also makes it possible to keep track of and locate references to the different instances of microservices.

RPC: an interaction style where remote calls resembles procedure call in a local setting. In particular, we have chosen to use gRPC and adopt its philosophy, centered around “coarse-grained message exchange while avoiding the pitfalls of distributed objects and the fallacies of ignoring the network” [5], therefore avoiding many of the issues normally stemming from RPC/RMI. Additionally, its support out-of-the-box for many languages, load-balancing, blocking & non-blocking support and HTTP transport make it a perfect fit for use in a microservice architecture.

Pub-sub/Message queuing: a messaging pattern which allows to obtain a high degree of decoupling among components in both space and time: a publisher does not need to know of the existence of a subscriber and can continue to operate independently even if there is none. By doing so, asynchronous communication is implemented.

Tiered architecture: the separation of the architecture in layers, such as presentation, business and data, offers great flexibility, maintainability and scalability. In particular, by separating the business from the data and having communication between components always go through a predefined API, we do not have to worry about each knowing the internal representation of the others.

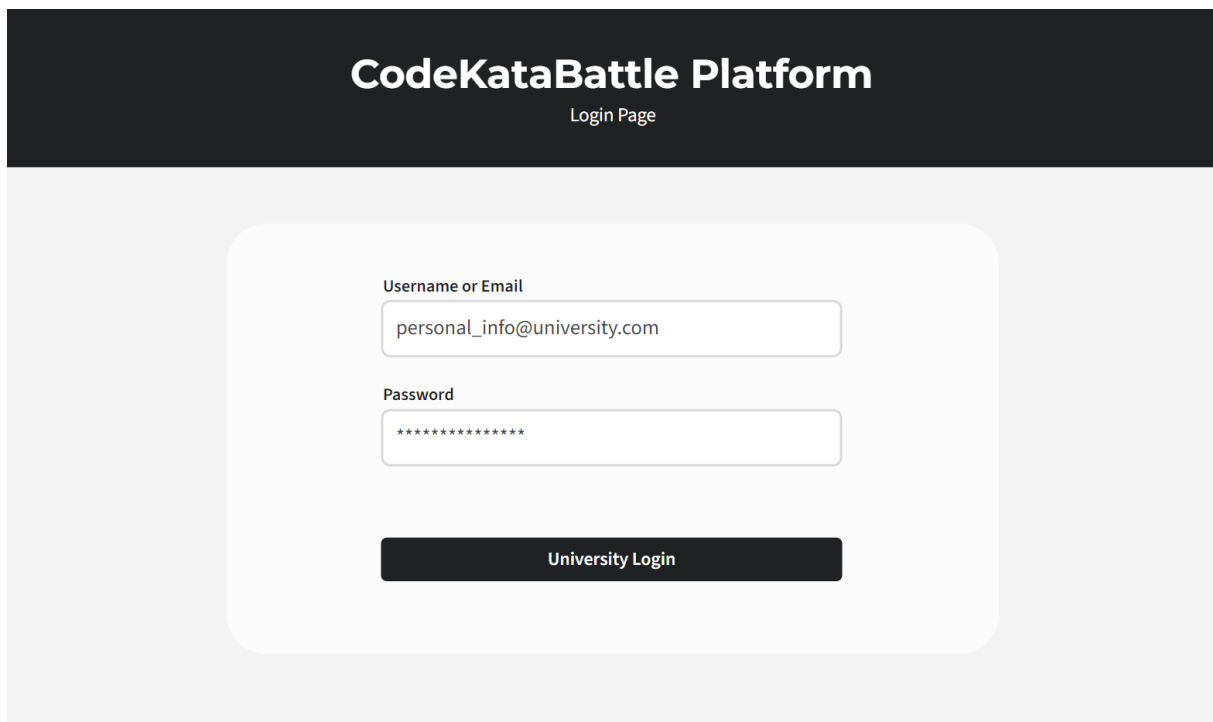
Stateless components: the use of stateless components, which do not have memory of interactions they have previously received, allows to decouple successive calls to services and facilitates the horizontal scaling of the system.

2.7. Other design decisions

Relational DBMS: a relational database is the traditional choice because of their guarantees given by ACID properties enforcing consistency on the data, as well as their fixed schema with well-defined entities which fits well with the well-defined model we have identified.

3 | User Interface Design

Depending on how communication is implemented between the CKB Platform user service and the university's user management system, this login screen may allow direct login by interfacing with the university's login or it will simply redirect to the university's login site by subsequently logging in.



The image shows a login page for the CodeKataBattle Platform. The page has a dark header with the title "CodeKataBattle Platform" and the subtitle "Login Page". Below the header is a light gray background. In the center, there is a white rounded rectangle containing the login form. The form has two input fields: "Username or Email" with the value "personal_info@university.com" and "Password" with masked characters "*****". Below the input fields is a dark button labeled "University Login".

Figure 3.1: Login

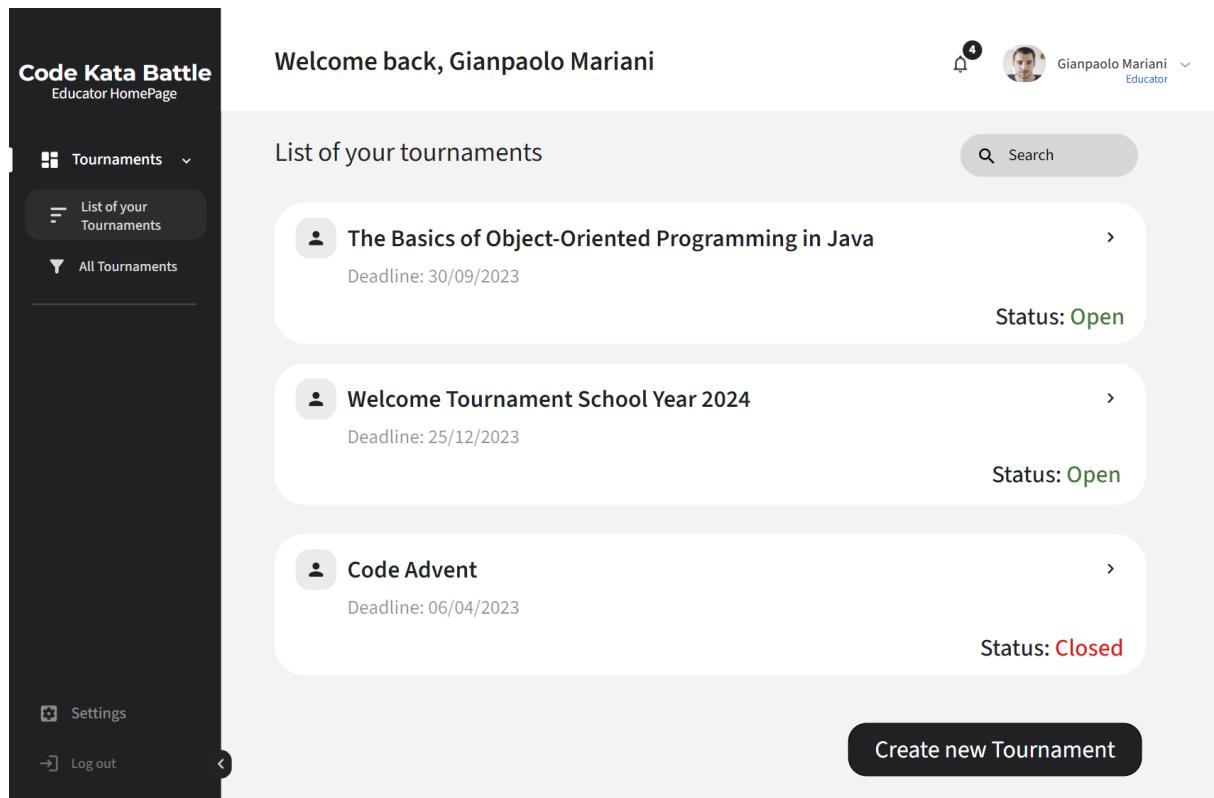


Figure 3.2: Home Page Educator

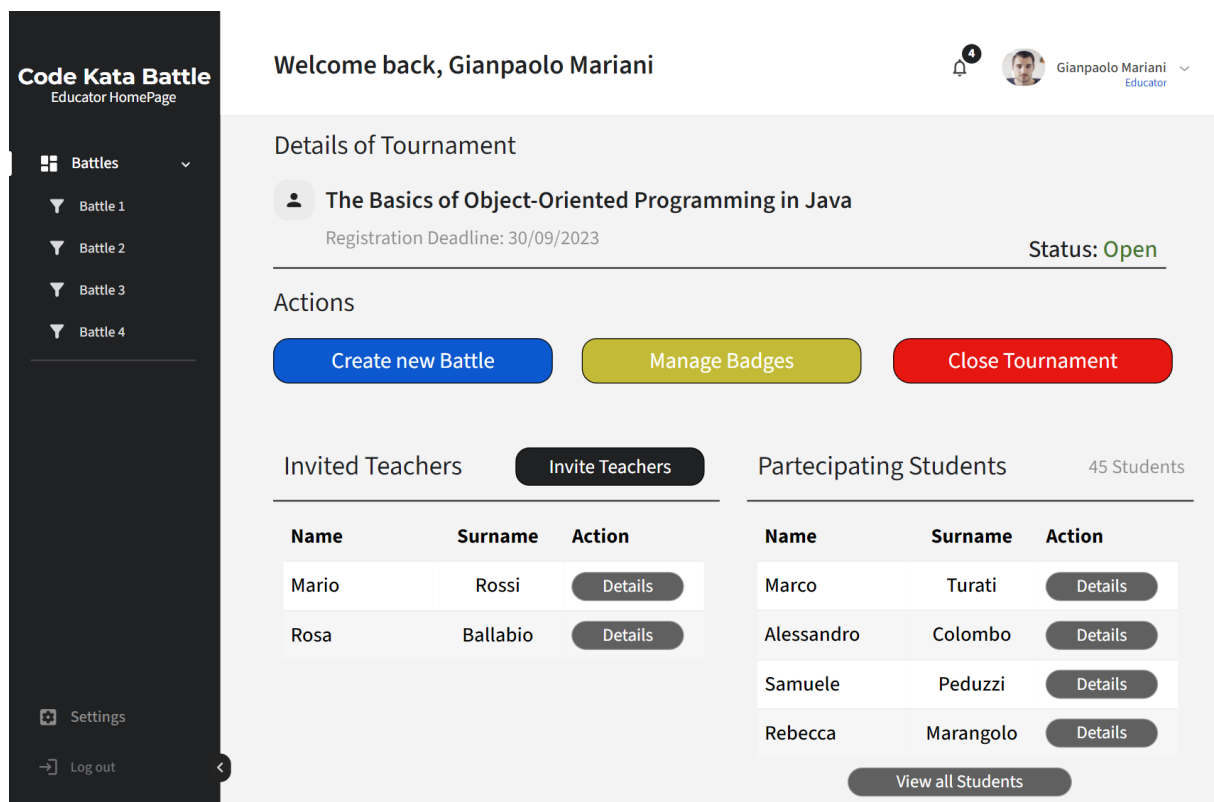


Figure 3.3: Details Tournament Educator

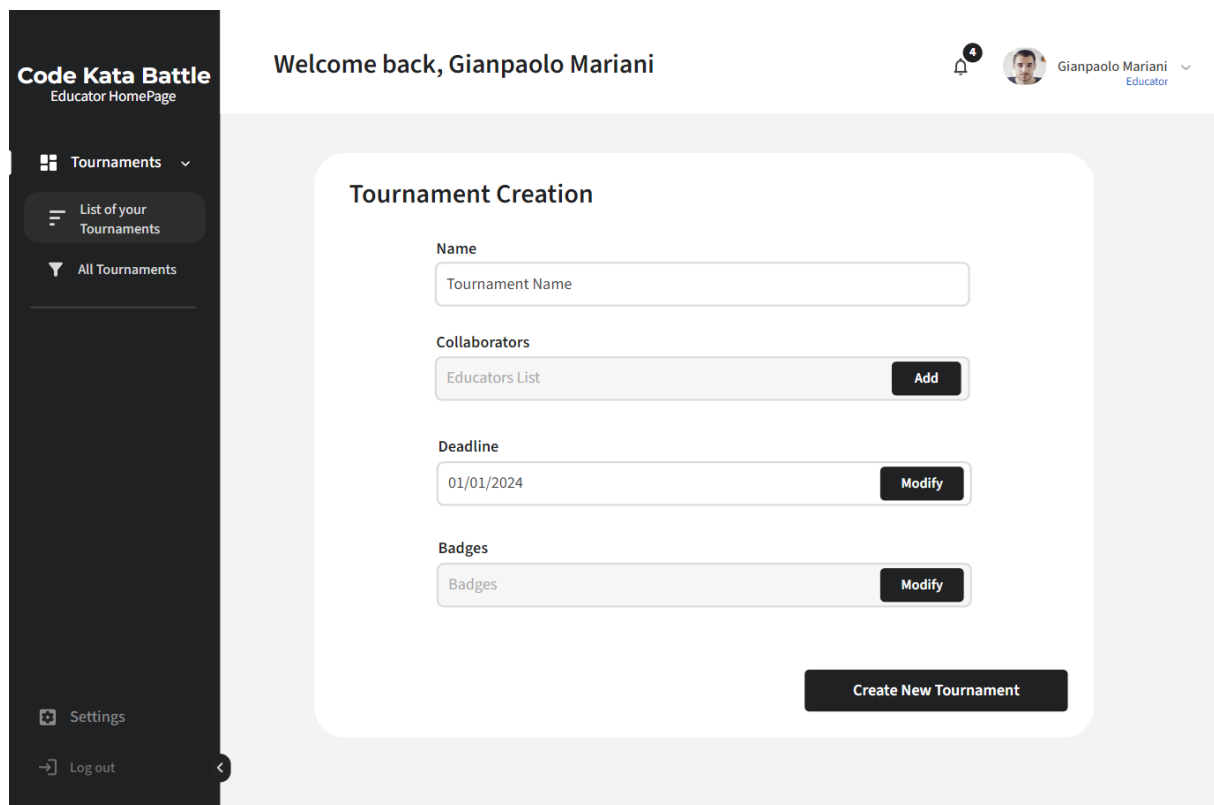


Figure 3.4: Details Tournament Educator

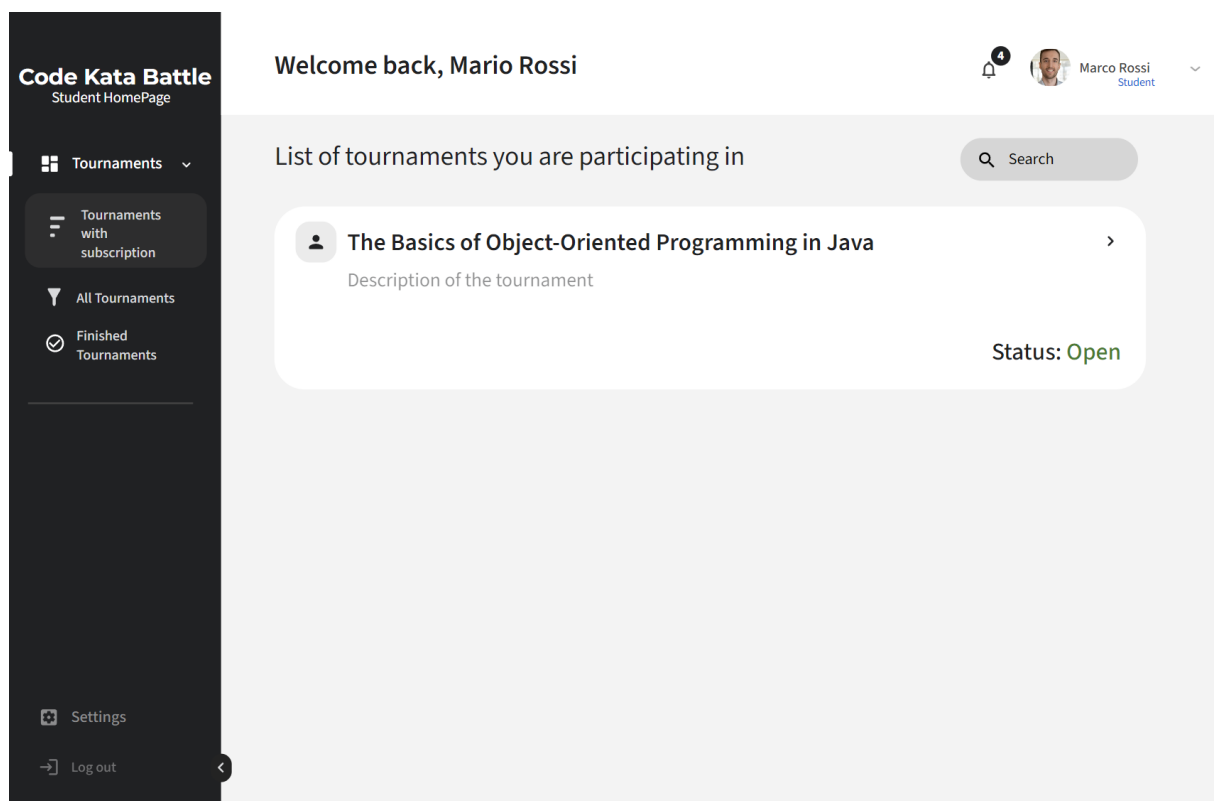


Figure 3.5: HomePage Student

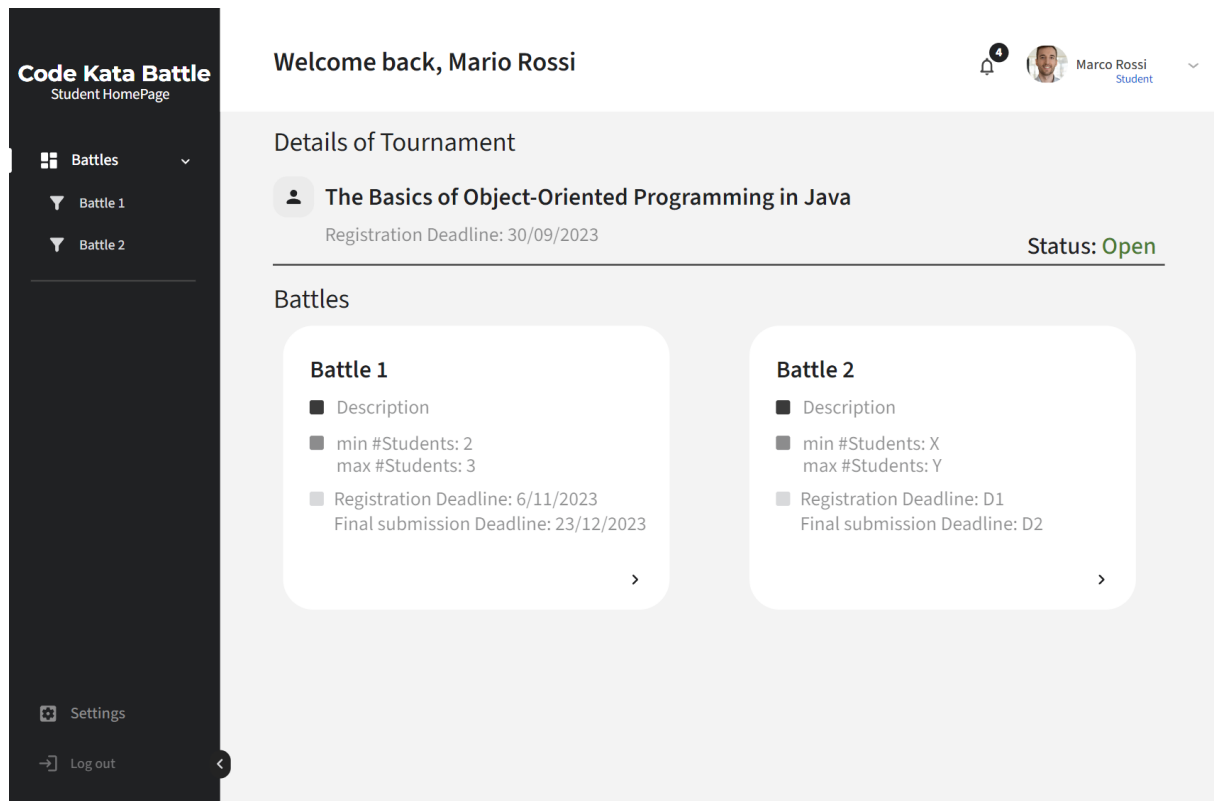


Figure 3.6: Details Tournament Student

4 | Requirements Treceability

In this section the requirements specified in the RASD are mapped to the components defined above. For brevity, the API Gateway is omitted as, since it mediates most of the operations, it would need to be specified in almost all requirements.

	Description	Components
R1	The system shall allow users to log in with SSO.	User Service
R2	The system shall allow the educator to create a tournament.	Platform Service
R2.1	The system shall allow the educator to specify the subscription deadline of a tournament.	Platform Service
R2.2	The system shall allow the educator to grant other colleagues the permission to create battles within the context of a specific tournament.	User Service, Platform Service
R2.3	The system shall notify the students of the new tournament.	Platform Service, Notification Service
R2.4	The system shall allow the educator who create the tournament to close it.	Platform Service
R3	The system shall allow students to subscribe to a tournament.	Platform Service
R4	The system shall allow the educator to create a battle within the context of a specific tournament.	Platform Service
R4.1	The system shall allow the educator to set minimum and maximum number of students per group.	Platform Service
R4.2	The system shall allow the educator to set a textual description.	Platform Service
R4.3	The system shall allow the educator to set the programming language.	Platform Service
R4.4	The system shall allow the educator to upload a set of test cases.	Platform Service, Build and Test Service
R4.5	The system shall allow the educator to upload a build automation script.	Platform Service, Build and Test Service

	Description	Components
R4.6	The system shall allow the educator to set a registration deadline.	Platform Service
R4.7	The system shall allow the educator to set a final submission deadline.	Platform Service
R4.8	The system shall allow the educator to enable manual evaluation.	Platform Service
R4.9	The system shall allow the educator to select aspects that should be evaluated by the static analysis tool, such as security, reliability, and maintainability.	Platform Service, Static Analysis Service
R4.10	The system shall notify students subscribed to a tournament of the creation of a new battle.	Platform Service, Notification Service
R5	The system shall allow students to join a battle, respecting the minimum and maximum number of students per group.	Platform Service
R5.1	The system shall allow students to invite other students to join a battle.	Platform Service, User Service
R5.2	The system shall allow students to accept received invitation.	Platform Service
R6	When the registration deadline expires, the system shall create a GitHub repository containing the code kata.	Platform Service
R7	The system shall send the link to all students who are members of subscribed teams.	Notification Service
R8	The system shall expose an API that can be called by the GitHub Action platform.	API Gateway
R9	On each push, the system shall calculate and update the battle score of the team.	Platform Service, Build and Test Service, Static Analysis Service
R9.1	The system shall pull the latest sources.	Build and Test Service
R9.2	The system shall analyze quality level of the sources, based on the aspect selected by the educator.	Static Analysis Service
R9.3	The system shall run tests uploaded by the educator.	Build and Test Service
R9.4	The system shall measure the time passed between the registration deadline and the last commit.	Platform Service

	Description	Components
R10	The system shall allow students and educators involved in the battle to see the current rank evolving during the battle.	Platform Service
R11	When the submission deadline expires, if manual evaluation is required, the system shall change the state of the battle to the consolidation stage.	Platform Service
R12	When the submission deadline expires, if manual evaluation is not required, the system shall close the battle.	Platform Service
R13	During the consolidation stage, if manual evaluation is required, the system shall allow the educator to go through the sources produced by each team to assign his/her score.	Platform Service
R14	At the end of a battle, the system shall calculate the final rank.	Platform Service
R15	When the final rank is available, the system shall notify all students participating in the battle.	Platform Service, Notification Service
R16	At the end of each battle, the platform updates the personal tournament score of each student, that is the sum of all battle scores received in that tournament.	Platform Service
R17	The system shall allow users to see the tournament ranks.	Platform Service
R18	At the end of a tournament, the system shall calculate the final tournament rank.	Platform Service
R19	When the final tournament rank is available, the system shall notify all students involved in the tournament.	Platform Service, Notification Service
R20	The system shall allow the educator who create a tournament, to create badges within the context of the tournament.	Platform Service, Badge Service
R20.1	The system shall allow the educator to specify the title of the badge.	Badge Service
R20.2	The system shall allow the educator to create a new variable that represent any piece of information available in the platform relevant for scoring.	Badge Service

	Description	Components
R20.3	The system shall allow the educator to specify one or more rules that must be fulfilled to achieve the badge, based on variables.	Badge Service
R20.4	The system shall allow users to visualize badges collected by a student.	Badge Service
R21	At the end of a tournament, the system shall assign badges to the student who fulfilled the rules.	Badge Service

Table 4.1: Components traceability matrix

5 | Implementation, Integration and Test Plan

The chosen architecture guarantees high decoupling between components, so that most of them can be developed and unit tested completely independently, without the need to mock other components. Once those are developed, we can integrate them at the end and do additional integration testing. We can identify a few different types components:

Independent components: these are back-end components that can be developed fully independently from one another and do not need to directly integrate with any other service

External components: these are the components provided by third-parties, which are supposed to be already reliable, but may need to be stubbed when unit testing our own services

Integrating components: these are the back-end components whose sole role is of allowing communication between other services and therefore need to be directly integrated with those

Front-end components: these are the presentational components that belong to the client layer which rely on the back-end REST API, they can therefore be unit tested by mocking it

In order to visualize how much each component will need to be tested and plan accordingly, we can use the following table, where we associate to each piece of functionality the difficulty of its implementation and its importance for the final user experience:

Feature	Importance	Difficulty
Login	High	Medium
Tournament and Battle Management	High	Low
Tournament and Battle Participation	High	Medium
Automatic evaluation	High	High
Manual evaluation	High	Low
Tournament rankings	Medium	Low
Battle rankings	Medium	Low
Notifications	Medium	High
Badges Management	Low	Medium
Badges Assignment	Low	High
Badges Visualization	Low	High

Table 5.1: Importance and difficulty of features

5.1. Development and Test Plan

All the components will be developed and tested using a bottom-up approach, in order to reduce as much as possible stubs and mocks, which would add additional overhead to the development. All the independent microservices can be developed first and in parallel, prioritizing components of high importance which need to be tested more thoroughly as outlined by the table above. Development can also start on front-end components, which can mock the REST API during testing. Lastly, integrating components such as the API Gateway can be developed, after which everything can be integration tested.

After that is completed, system and e2e testing should be done to verify the adherence to the specified requirements. This phase also include:

- Performance testing, to detect bottlenecks and test the scaling strategies.
- Load testing, to identify memory leaks, overflows and other memory-related problems.
- Stress testing: to make sure that the system recovers gracefully after failures.

Once a beta version of the software is available, it's possible to perform some acceptance testing which should involve different users, including stakeholders, to test if the system responds to actual needs and constraints.

5.2. Components integration

Here components and subsystems are illustrated via graphs.

Subsystems are a group of components meant to be integration tested together after unit testing.

Integration testing is carried out in a bottom-up approach, so drivers must be implemented to test the different independent subsystems as each subsystem is developed.

The first components that can be tested together are the Notification Service and RabbitMQ. That's because the subsystem is implementing medium importance features and thus proper testing with the queue needs to be done carefully.

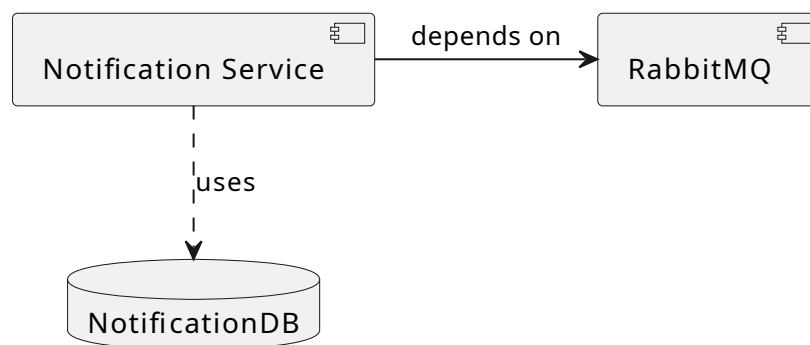


Figure 5.1: Notification subsystem

Then the Platform subsystem can be integrated with the Queue of the Notification subsystem. Obtaining the Platform and Notification Subsystem.

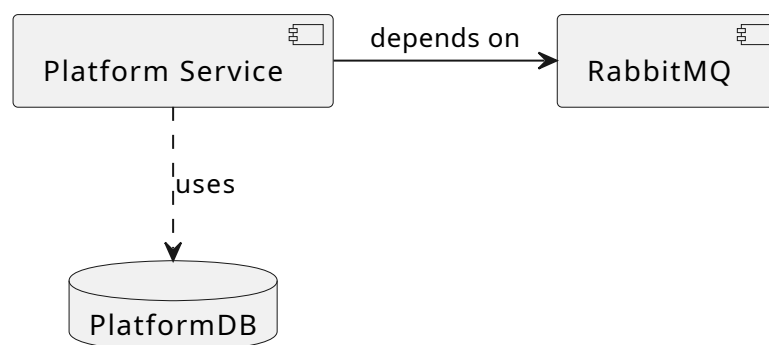


Figure 5.2: Platform and Notification subsystem

The following four subsystems might be developed in any order, or simultaneously, as they do not depend on each other. It is recommended to develop first the subsystems that implement High Importance features, such as:

the User subsystem, the Build and Test subsystem, and the Static Analysis subsystem, and then the Badge subsystem that implements Low Importance functionality.

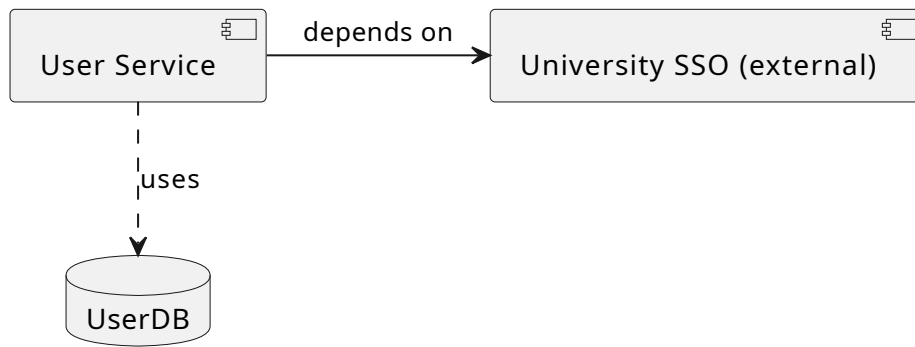


Figure 5.3: User subsystem

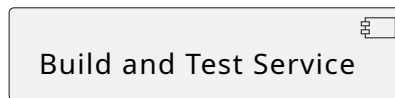


Figure 5.4: Build and Test subsystem

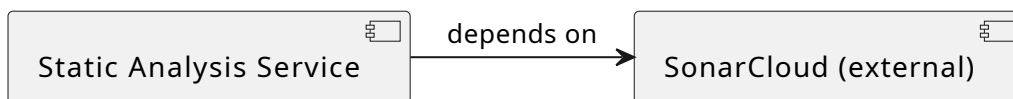


Figure 5.5: Static Analysis subsystem



Figure 5.6: Badge subsystem

After that we can also develop the Website CDN.



Figure 5.7: Website CDN subsystem

At this point all subsystems have been implemented and tested, so the next step is performing integration testing between the subsystems and the API Gateway.

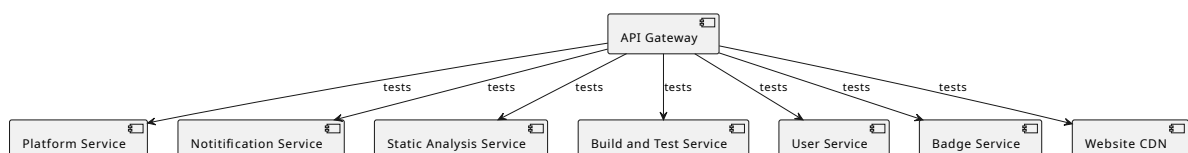


Figure 5.8: API Gateway subsystem

The integration of all the different subsystems with the Gateway API is a critical operation and performing it all at once will likely lead to errors (because of the big-bang like approach). To avoid this, the integration can be broken down into smaller integrations by continuing to follow the bottom-up approach.

We can proceed to integrate the API Gateway with the Platform and Notification subsystem.

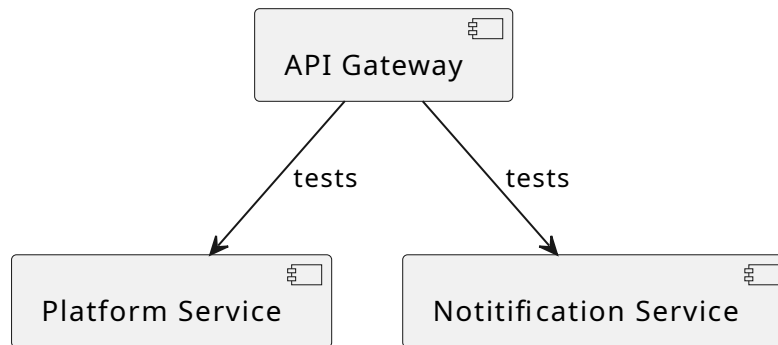


Figure 5.9: API Gateway subsystem step 1

Then, it's possible to integrate the API Gateway with the User and the Badge subsystems.

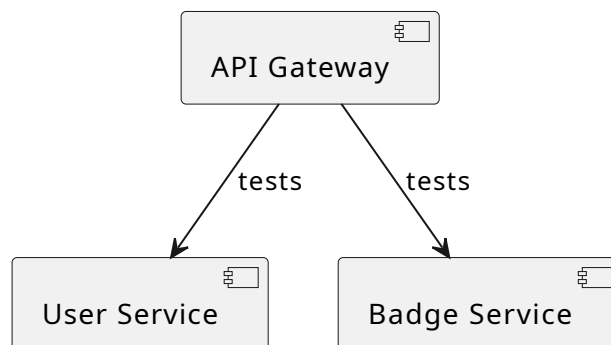


Figure 5.10: API Gateway subsystem step 2

After that, integrate the API Gateway with the Static Analysis and Build and Test subsystems.

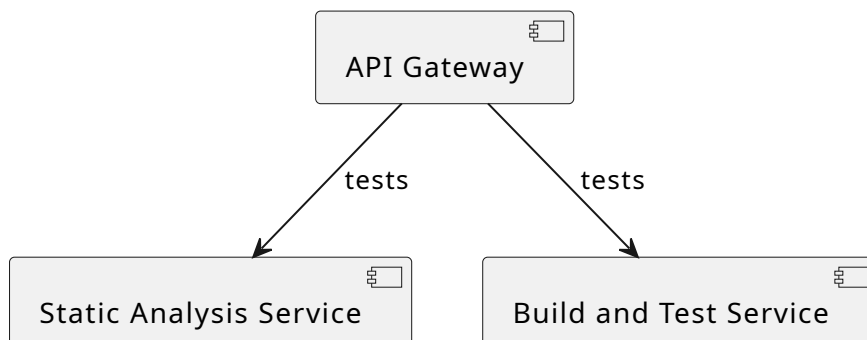


Figure 5.11: API Gateway subsystem step 3

Finally, when all is integrated and tested successfully we can proceed to integrate it with the Website CDN.

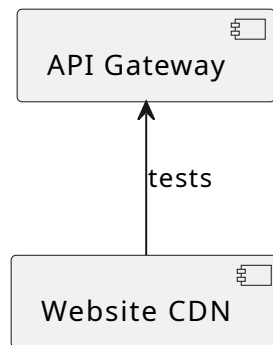


Figure 5.12: API Gateway subsystem step 4

6 | Effort Spent

Member of group	Chapter	Time spent
Giacomo Orsenigo	Introduction	1h
	Architectural Design	11.5h
	User Interface Design	
	Requirements Traceability	1h
	Implementation, Integration and Test Plan	1h
Francesco Ferlin	Introduction	1h
	Architectural Design	10.5h
	User Interface Design	
	Requirements Traceability	1h
	Implementation, Integration and Test Plan	2h
Federico Sacconi	Introduction	1.5h
	Architectural Design	8h
	User Interface Design	2h
	Requirements Traceability	1h
	Implementation, Integration and Test Plan	4h

Table 6.1: Time spent by each member of group.

Bibliography

- [1] Amazon EC2. URL <https://aws.amazon.com/it/ec2/>.
- [2] Amazon ELB. URL <https://aws.amazon.com/it/elasticloadbalancing/>.
- [3] Amazon VPC. URL <https://aws.amazon.com/it/vpc/>.
- [4] gRPC. URL <https://grpc.io/>.
- [5] gRPC Principles & Requirements. URL <https://grpc.io/blog/principles/#services-not-objects-messages-not-references>.
- [6] Jenkins. URL <https://www.jenkins.io/>.
- [7] Jenkins scaling architecture. URL <https://www.jenkins.io/doc/book/scaling/architecting-for-scale/>.
- [8] Project Reactor. URL <https://projectreactor.io/>.
- [9] RabbitMQ. URL <https://rabbitmq.com/>.
- [10] SonarCloud GitHub. URL https://next.sonarqube.com/sonarqube/web_api/api/alm_integrations/import_github_project.
- [11] SonarCloud Quality Profile. URL https://next.sonarqube.com/sonarqube/web_api/api/qualityprofiles/add_project/.
- [12] SonarCloud WebHooks. URL <https://docs.sonarsource.com/sonarqube/latest/project-administration/webhooks/>.
- [13] Spring AMQP. URL <https://spring.io/projects/spring-amqp/>.
- [14] Spring Boot. URL <https://spring.io/projects/spring-boot/>.
- [15] Spring Cloud Gateway. URL <https://spring.io/projects/spring-cloud-gateway/>.
- [16] Web Push API on MDN. URL https://developer.mozilla.org/en-US/docs/Web/API/Push_API.

List of Figures

2.1	Abstract System View	5
2.2	Component view diagram	6
2.3	Platform ER	8
2.4	Badges ER	9
2.5	Notification ER	9
2.6	Deployment view diagram	10
2.7	Login	12
2.8	User gets list of tournaments	13
2.9	User gets tournament details	13
2.10	User gets battle details	14
2.11	Notification is sent	15
2.12	Educator creates a new Tournament	16
2.13	Educator creates a new badge	17
2.14	Educator creates a new Battle for an Existing Tournament	18
2.15	Student joins to an existing Tournament by receiving a notification	19
2.16	Students create a team for a tournament battle	21
2.17	Student forks the repository	22
2.18	Student pushes and triggers automatic evaluation	23
2.19	Educator manually evaluates teams	24
2.20	Educator closes a tournament	25
3.1	Login	33
3.2	HomePage Educator	34
3.3	Details Tournament Educator	34
3.4	Details Tournament Educator	35
3.5	HomePage Student	35
3.6	Details Tournament Student	36
5.1	Notitification subsystem	43
5.2	Platform and Notification subsystem	43
5.3	User subsystem	44
5.4	Build and Test subsystem	44
5.5	Static Analysis subsystem	44
5.6	Badge subsystem	44
5.7	Website CDN subsystem	44
5.8	API Gateway subsystem	44
5.9	API Gateway subsystem step 1	45
5.10	API Gateway subsystem step 2	45

5.11 API Gateway subsystem step 3	45
5.12 API Gateway subsystem step 4	46

List of Tables

4.1	Components traceability matrix	40
5.1	Importance and difficulty of features	42
6.1	Time spent by each member of group.	47